Xic Reference Manual

With OpenAccess Support

Whiteley Research Incorporated
Sunnyvale, CA 94086

Release 4.3.14
February 9, 2025

Xic is part of the *XicTools* software package for integrated circuit design. *Xic* was primarily authored by S. R. Whiteley, Whiteley Research inc., Sunnyvale CA USA.

This page intentionally left blank.

Contents

This page intentionally left blank.

Chapter 1

Introduction to *Xic*

This chapter will provide an overview of the *Xic* program, setup and initialization information, and information for basic use. Detailed information on the various commands, features, and modes will be found in the following chapters. Information on file formats and other rather technical topics can be found in the appendices. New users should read this chapter and the first two sections of the following chapter thoroughly, and read the sections in the remaining chapters describing the commands referred to in the usage sections in chapter 2. The on-line help contains most of the information presented in this manual, in a cross-referenced format. Users will likely make extensive use of the help system. The information provided in the help system is generally more up-to-date than can be provided in the manual, and should be considered to be correct if there is ever a conflict.

Whiteley Research is more than happy to assist users by answering questions and providing information. The "**WR**" button in the *Xic* interface brings up a mail client which can be used to send questions to Whiteley Research, which will be answered as soon as possible. However, in order for this service to operate efficiently, it is requested that users make an effort to answer questions by reading the provided documentation before contacting Whiteley Research.

In this manual, text which is provided in `typewriter` font represents verbatim input to or output from the program. Text enclosed in square brackets ([text]) is optional in the given context, as in optional command arguments, whereas other text should be provided as indicated. Text which is *italicized* should be replaced with the necessary input, as described in the accompanying text.

1.1 *Xic* Graphical Editor Overview

Xic is a dual-mode graphical editing tool. In the physical editing mode, *Xic* is a hierarchical mask layout editor, with interactive and batch mode design rule checking, arbitrary angle polygon and wire support, netlist and parameter value extraction, and many more advanced features. In electrical layout mode, *Xic* serves as a hierarchical electrical schematic editor and schematic capture front end for SPICE. In the *XicTools* environment, circuit simulation can be performed and results analyzed from within *Xic*, through an interprocess communication channel established to the companion *WRspice* program.

Arrayed along the top of the main window is a toolbar containing drop-down menu selectors. Below the menu bar is a tool bar containing buttons and other controls, including the coordinate readout area to the right. To the left of the main window is an array of additional command buttons. These menu commands control the operation of *Xic*. The main drawing window occupies the largest section of the main window. The main drawing window supports drag and drop as a drop receiver for files. To the left of the main drawing window is the layer menu, which displays a listing of the layers used in the process. The layers, and their attributes, are specified in a technology file read by *Xic* at program startup.

Just below the main drawing window is the prompt line, which provides a channel for text-mode interaction with the program. In the same row, below the buttons in the side menu is the key press buffer area, which records characters typed into the graphics window. It is invisible until characters are typed. The typed characters are interpreted as command accelerators. Below the prompt line, at the bottom of the main window, is a status line which provides information about the current program operating state.

The WR button, in the upper left corner of the main window, brings up a mail client which can be used to send messages and files via internet mail. It is preloaded with the address of the technical support group at Whiteley Research.

Despite the array of features, *Xic* is intended to be straightforward and intuitive to use, *Xic* has extensive on-line documentation available through a context-sensitive help system. This help system can easily be augmented and customized by the user, so that the user's design rules and tips, and other technical information can be made available from within *Xic*.

Xic includes a native, script execution facility, with plug-in support for Python and Tcl/Tk. These languages will be available if installed on the user's computer. The native scripting language is a straightforward but powerful C-like language with a rich library of primitives for controlling the operation of *Xic*. Scripting can be used for automation, for parameterized cells and executable labels, and to implement user-defined commands. These commands may appear as buttons in the **User Menu**.

One application of the user scripts is to provide simple, menu based commands for creating geometrical objects, devices, or parameterized device structures for use in circuit layout. Further uses for this capability are limited only by the user's imagination.

Xic can execute scripts in batch and server modes, allowing geometrical manipulations to be performed in a background or non-local environment. As a server, *Xic* can serve as the workhorse back-end for web-based or turn-key third-party products or services, or in-house custom applications.

Xic provides access to the OpenAccess database via a plug-in. It can utilize the OpenAccess database provided with Cadence Virtuoso, or Synopsys/Ciranova PyCell Studio, and others. *Xic* has some limited compatibility with Cadence Virtuoso: *Xic* can directly read Cadence technology and display resource files, and can read layout and some schematic and schematic symbol views.

Default schematic editing support is provided for a wide variety of devices, even Josephson junctions. Additional devices and subcircuits can easily be added by the user, or changes can be made to existing devices, by editing a single text file. *Xic* also provides a high-powered model library search engine compatible with any SPICE format model or subcircuit library files, such as those provided by semiconductor manufacturers.

Hard copy support is available for a variety of printers and file formats, including PostScript (mono and color), HPGL, and HP laser.

Xic has support for several archive layout file formats, plus native input and output. Data input in a given format will remain in that format, unless explicitly converted.

Xic produces data files which contain both electrical and physical data, though one of these two data areas may be empty. The file format used can be one of:

- The native format, in which each cell of a design is written to an independent ASCII file.
- An extension of GDSII, a binary format where the entire design can be written to a single file.

- The newer and more compact OASIS format, which is a replacement for GDSII.
- An extension of CIF, a multi-cell format, somewhat archaic, but an ASCII format so human-readable.
- The CGX file format, developed by Whiteley Research.
- OpenAccess, a third-party database used by Cadence and others.

Xic will read any of these file types automatically, and save any editing changes in the same file type unless instructed otherwise.

Built-in converters can be used to convert between the file formats. It is possible to "strip" the output, providing a physical-data file completely compatible with the industry standard file formats, for portability of mask layout information. It is also possible to read and write a "text-mode" version of GDSII files, which can be used to repair corrupted or misbehaving GDSII databases.

Xic provides a powerful facility for translating between supported layout file formats, while potentially modifying the data. Possible modifications include layer filtering and aliasing, cell name global modification and aliasing, flattening, and spatial filtering to a rectangular area with or without clipping, cell replacement, and more. These operations can be applied to very large files, as a unique technique minimizes memory use.

In physical mode, design rule checking can be performed as each new object is created of modified. Batch mode checking is also available, either in the foreground, or as a background child process. The philosophy of *Xic* is that it is never in the user's best interest to "cheat" in the enforcement of design rules, yet there may be times when a given rule is not appropriate, and a modified rule should be used. Following this philosophy, the user is given complete control over the design rules, which can be edited, disabled, or rules added interactively. The user can initiate batch mode design rule checking over a given area or over a complete cell. Design rule checking is performed over a pseudo-flat internal representation of the layout, so that physical rules are checked without any constraint based upon which subcells contain the geometry.

Xic has provision for netlist and parameter extraction. The netlist obtained from the physical layout, plus extracted physical device parameters, can be used to generate a SPICE output file, and even a schematic. Automated layout vs. schematic (LVS) testing is provided.

1.2 *Xic* Feature Sets

The *Xic* user may have access to only a subset of features. These feature sets correspond to "virtual" products, that were historically separate programs.

There are three feature sets available. The "`FULL`" set enables all *Xic* features. The "`EDITOR`" feature set corresponds to the *XicII* program, which provides physical layout editing capability. The "`VIEWER`" feature set corresponds to the *Xiv* program, which allows physical layout viewing. The subsections that follow describe these feature sets in more detail.

1.2.1 The EDITOR Feature Set

This feature set corresponds to the *XicII* virtual product. This was once a stand-alone layout editor product. Currently, the same functionality is provided via running *Xic* with the EDITOR feature set, which was formerly imposed during license authentication.

One can force running with the EDITOR feature set by setting the environment variable FORCE_XICII before starting the *Xic* program.

This feature set restricts the functionality to physical layout editing. This provides a low-cost alternative for users that do not require the full functionality of *Xic*. We will continue to use "*XicII*" to refer to *Xic* running with this feature set.

In order to streamline support and maintenance, the documentation tree, i.e., the manual, help database, and release notes, is common to all feature sets. This is a slight disadvantage to users of restricted feature sets, as the documentation contains descriptions of disabled features, which may lead to confusion. However, this greatly simplifies maintaining the documentation.

This section will list the differences and features that are unavailable in the *XicII* virtual product.

1. **Technology File**
   Parts of the technology file that relate to features that are not available in *XicII* are ignored, but will generate warning messages. In the example technology files, these features are enclosed in macro-tested blocks to avoid the warnings. The syntax is

   ```
   If FEATURESET == "FULL"
   ...
   EndIf
   ```

   The right side of the conditional can take these values:

   `"FULL"`
   All features enabled.

   `"EDITOR"`
   Layout editing feature set (*XicII*)

   `"VIEWER"`
   Layout viewing feature set (*Xiv*)

2. **No Design Rule Checking**
   XicII does not have DRC support, consequently there is no **DRC Menu** in *XicII*.

3. **No Electrical Mode**
   XicII is a physical layout tool only. There is no schematic entry, and no SPICE capability. There is no **Electrical** or **Physical** button in the **View Menu**.

4. **No Extraction**
   XicII has no extraction capability and no **Extract Menu**.

5. **No Batch or Server Modes**
   The background processing capability is not available in *XicII*.

6. **'!' Commands**
   The '!' commands in *XicII* are identical to those in *Xic*, however '!' commands in *XicII* which relate to unavailable features will not be recognized.

1.2.2 The VIEWER Feature Set

This feature set corresponds to the *Xiv* virtual product. This was once a stand-alone layout viewer product. Currently, the same functionality is provided via running *Xic* with the VIEWER feature set, which was formerly imposed during license authentication.

One can force running with the VIEWER feature set by setting the environment variable FORCE_XIV before starting the *Xic* program.

This feature set restricts the functionality to physical layout viewing. This provides a low-cost alternative for users that do not require the full functionality of *Xic*. We will continue to use "*Xiv*" to refer to *Xic* running with this feature set.

In order to streamline support and maintenance, the documentation tree, i.e., the manual, help database, and release notes, is common to all feature sets. This is a slight disadvantage to users of restricted feature sets, as the documentation contains descriptions of disabled features, which may lead to confusion. However, this greatly simplifies maintaining the documentation.

This section will list the differences and features that are unavailable in the *Xiv* virtual product.

1. **Technology File**
   Parts of the technology file that relate to features that are not available in *Xiv* are ignored, but will generate warning messages. In the example technology files, these features are enclosed in macro-tested blocks to avoid the warnings. The syntax is

   ```
   If FEATURESET == "FULL"
   ...
   EndIf
   ```

   The right side of the conditional can take these values:

   `"FULL"`
   All features enabled.

   `"EDITOR"`
   Layout editing feature set (*XicII*)

   `"VIEWER"`
   Layout viewing feature set (*Xiv*)

2. **No Editing**
   All cells are treated as read-only. The menus that relate to changing the layout (**Edit** and **Modify**) are absent.

3. **No Design Rule Checking**
   Xiv does not have DRC support, consequently the **DRC Menu** is absent.

4. **No Electrical Mode**
   Xiv is a physical layout viewing tool only. There is no schematic entry, and no SPICE capability. There is no **Electrical** or **Physical** button in the **View Menu**.

5. **No Extraction**
   Xiv has no extraction capability and no **Extract Menu**.

6. **No User Menu**
   Scripting is not available.

7. **No Batch or Server Modes**
   The background processing capability is not available in *Xiv*.

8. **'!' Commands**
   The '!' commands in *Xiv* are identical to those in *Xic*, however '!' commands in *Xiv* which relate to unavailable features will not be recognized.

1.3 A Quick Tour of *Xic* Capabilities

1.3.1 History of *Xic*

The precursor to *Xic* was the Kic layout editor, a very simple no-frills layout editor developed at Berkeley in the 1980's. In the late 1980s, the author needed a layout editor to support contract development and research efforts in superconductive electronics, and adopted Kic, run under something called a "DOS extender" (to support 32-bit applications) on an early and very expensive i386 computer. This required extensive modification to Kic, mostly to support the PC graphics. Kic is still available as free software on the Whiteley Research web site.

After Unix became available for 386/486 PCs in the form of the FreeBSD operating system, DOS and direct-write graphics became history. *Xic* became a separate program in late 1995, initially using the X-window system (Xt) user interface toolkit. Over the following years, *Xic* became a full-time development effort, and the extraction, DRC, and other subsystems were added. Although to this day faint similarities to Kic exist, internally the code has been replaced has been replaced by several iterations of more modern code, and the database and other systems were replaced with improved implementations.

Eventually, *Xic* underwent a complete rewrite into C++ (from C) to improve maintainability and organization. The GTK toolkit was adopted for the user interface.

Whiteley Research Inc. was founded in 1996 to market *Xic*, and the companion *WRspice* program. Since then, *Xic* has continued to develop, as new users brought forward new ideas and requirements.

1.3.2 General

Xic provides a menu of buttons along the side (the "side menu"), and s number of drop-down menus along the top of the main window. *Xic* responds to key presses in various ways, and provides an input/output text area just below the main window. Key presses are interpreted as macros, special commands, menu command accelerators, or as '!' commands. Several control sequences directly initiate certain operations, for example **Ctrl-r** will redraw the window and **Ctrl-g** will prompt for grid parameters. Other control sequences will trigger menu commands as accelerators, and typing the unique prefix of the command name (as shown in the tool tip which appears as the mouse pointer hovers over a menu entry) will trigger menu commands. If '!' is pressed, the rest of the sequence (until **Enter** is pressed) is taken as an internal or Unix shell command. If '?' is pressed, the rest of the sequence (until **Enter** is pressed) is taken as a help database keyword.

1.3.3 The Help System

Xic contains a comprehensive HTML-based on-line help system. The help viewer can also function as a web browser, providing access to internet resources. The viewer can serve as an input device for scripts, i.e., the window would contain a form which provides parameters to a script. The help database can be augmented by the user, allowing local information to be easily accessed.

Xic is internet aware, and can actually open design files served by a remote HTTP or FTP host. Files can also be opened in response to clicking on links in the help viewer.

1.3.4 Cadence Virtuoso and OpenAccess Compatibility

Xic can read and write design data to an OpenAccess database, but OpenAccess is not required. *Xic* can read and use ASCII technology and DRF files intended for Cadence Virtuoso and other similar tools, as provided by chip foundries. *Xic* can read schematic, symbol, and layout views produced by Cadence, and to varying degrees, obtain a working, simulatable cell hierarchy. Presently, it is not possible to write back schematic information to Cadence without corruption.

Xic supports Ciranova/Synopsys portable Python-based parameterized cells, and provides support for abutment and stretch handles in native parameterized cells.

1.3.5 Layout Editing

First and foremost, *Xic* is an editor for integrated circuit mask layouts. Although, in large measure, the notion of mask layout from manual polygon placement has disappeared in modern electronics, having been replaced by automated cell place and route, there are still many instances where layout viewing and editing are essential. *Xic* is designed the make this task efficient and straightforward.

Xic makes use of a proprietary database technology which provides extremely fast access to spatially-keyed data. The database technology has changed several times over the life of the program, and the current database, though invisible to users, is an important achievement.

Xic has a complete set of features for creating, moving, transforming, and modifying geometrical features and subcells, with complete undo/redo capability. Most of these features are accessed from the side menu, and from the **Edit Menu** and **Modify Menu**. Basic mouse operations allow selection, and moving, copying, or stretching selected objects. The ability to create physical text or crude images (e.g., for company logos) is built in.

Xic operates on a cell hierarchy, and has commands to push and pop the editing context through the hierarchy, and to flatten the hierarchy to arbitrary depth.

Some releases of *Xic* are 32-bit applications, and as such have an inherent memory limitation of about 3Gb. *Xic* has internal memory management which is designed to use as much available virtual memory as possible. On a system with sufficient memory, 2-3 GB files can be read in for editing directly. In *Xic* releases compiled for 64-bits, there is no such memory limitation.

1.3.6 Input/Output

The technology-specific information used by *Xic* is maintained in a single human-readable file. Most of the parameters set by the technology file can be set or reset from within *Xic*, and an updated technology file can be easily generated.

Xic can read or write files in several formats. These include

GDSII
: The industry-standard binary data format.

OASIS
: A new standard intended to replace GDSII and is far more compact.

CIF
: An ancient ASCII data format, still in use occasionally.

CGX
: A more compact replacement for GDSII developed by Whiteley Research (and placed in the public domain). It still uses fixed-sized integers, so is not nearly as compact as OASIS, but is simple to generate and parse.

Native
: A CIF-like cell-per-file format.

OpenAccess
: If present, *Xic* can read and write to an OpenAccess database, including the databases provided with Cadence Virtuoso and Ciranova PyCell Studio.

Any files in these formats can be read directly into *Xic*, whether or not the current technology matches. In fact, it is possible (and sometimes desirable) to start *Xic* with no technology file. As the file is read, *Xic* will add layers as necessary to represent the file. After changing layer colors and fill patterns as desired, a new technology file can be dumped.

Files can be read into the *Xic* database, and later written to disk in any of these formats. The default is to write in the same format as the original file.

In addition, format conversions can be applied directly, bypassing the database load. While converting, windowing operations (clipping), scaling, or flattening can be applied. Since *Xic* uses 64-bit file offsets, the direct conversions can be applied to huge files, even with 32-bit *Xic* binaries and modest memory.

1.3.7 Design Rule Checking

Xic has a built-in design rule checking engine, based on rules provided in the technology file or interactively in *Xic*. Both interactive (performed after every geometry modification) and batch-mode checking (foreground or background) is supported, in all or a portion of the layout.

Errors are reported in a log file, and indicators added on-screen. Clicking on the indicator can provide a close-up view of the error and explanatory text.

There is a rule editor that gives the user complete control over the rules and parameters in use. Although a fairly complete set of built-in tests is provided, user-defined tests allow more specialized tests to be performed. Special layers and flags allow objects and regions to be ignored during testing.

1.3.8 Electrical Mode

When *Xic* is in electrical mode (selectable under the **View Menu**) the main window is set up for schematic editing. A user-configurable palette of devices is available for placement. Devices are placed, wired together, and properties added to provide device parameters. Once a schematic is complete, it can be dumped as a SPICE file, or simulation can be performed interactively through the companion *WRspice* program. Performing a simulation is as easy as clicking the **run** button in the side menu, then, when complete, the **plot** button can be pressed, then clicking on nodes in the circuit diagram will display simulation plots. Plots can also be created while simulating, and are updated as the simulation progresses.

There are provisions for providing arbitrary names for nodes and devices in the circuit. The default is for *Xic* to define the names in most cases. There is a symbolic representation capability, enabling a subcircuit to have a special symbol, instead of a schematic, when used as a subcell.

Xic provides vectorized instance placements, and a complete net expression capability for multi-conductor wire net definition.

Electrical-mode data is “tied” to the physical mode data, and saved in the same file. This requires some extensions to be employed in the files. These extensions are 1) usually ignored by other programs, and 2) can be easily stripped out to ensure portability of physical data.

1.3.9 Extraction

The commands in the **Extract Menu** deal with the electrical/physical association defined for a cell, i.e., the electrical schematic and the physical layout.

It is not always necessary to enter the schematic by hand. A schematic can be produced from a SPICE file, or from the physical layout. The resulting schematic is perhaps not too useful from a human-readability standpoint, but is valid nonetheless. The user of course has the option to rearrange things and make other changes to promote readability and aesthetics.

There are provisions to update the schematic from the physical layout, either globally or per-device. It is possible to dump a netlist file or SPICE file created directly from the physical layout.

There is provision for LVS (layout vs. schematic) analysis.

The parameters that control extraction, and device definitions for extraction, generally appear in the technology file. These can be created or modified from within *Xic* through the technology parameter editor window.

1.3.10 Automation

Xic contains a just-in-time compiler for a powerful built-in scripting language. The native language is C-like, though a Lisp-like variant is also supported. There is also interoperability with the popular tcl/tk scripting language.

A lengthly and expanding set of interface functions allow *Xic* to be controlled by the scripts, and a very efficient computational geometry engine allows database manipulation.

Xic even supports a server mode, whereby *Xic* does not use graphics, and instead becomes a “daemon”, listening for job requests. Other applications can use the server for geometrical and other manipulations. A similar batch mode, where *Xic* again does not use graphics but instead executes a script and exits, is also available.

The user’s scripts can appear as command buttons in the **User Menu**, allowing custom operations to be easily accessible in normal operation.

The script language is used elsewhere, for example in user-defined design rule tests, and in executable labels. An executable label is a text object in a design that when clicked-on will perform some operation. Scripts are also used in template (parameterized) cells, which enable on-the-fly generation of subcells based on an arbitrary set of parameters.

1.4 A Quick Tour of the *Xic* Menus

1.4.1 Side Button Menu

Buttons arrayed along the side of the main window control the generation of objects - rectangles, polygons, wires (fixed-width paths), arcs, and rounded objects. Other buttons enable setting related defaults, such as wire end style and width, and the number of vertices used in "round" objects. Additional buttons control operations such as erase/yank/put, xor, clipping, and rotating. In electrical mode, this menu changes to provide buttons for adding connection terminals, controlling the node-naming, and managing the simulation interface to the companion *WRspice* program. These buttons are described in chapter 7.

1.4.2 Top Button Menu

There are a few buttons arrayed horizontally above the main drawing window, along with the coordinates display. These are associated with the layer table and selection control. The controls in this menu are described in 3.8.

The drop-down menus arrayed along the top of the main window control additional features.

In addition, there are a number of special '!' commands that are entered by typing the command name. These control or enable additional features that are not as frequently used.

Finally, there is a rather sophisticated scripting interface with a large collection of built-in functions, which enables the user to create automation scripts. These scripts can be initiated from the **User Menu**.

1.4.3 File Menu

The **File Menu** provides commands to open, save, and list files, cells, and other things. This menu also contains the printer interface.

File Menu			
Label	**Name**	**Pop-up**	**Function**
File Select	`fsel`	**File Selection**	Open file
Open	`open`	none	Open new cell or file
Save	`sv`	**Modified Cells**	Save modified cells
Save As	`save`	none	Save file, rename
Save As Device	`sadev`	**Device Parameters**	Electrical mode only, apply defaults and save device
Print	`hcopy`	**Print Control Panel**	Hard copy plot
Files List	`files`	**Path Files Listing**	List search path files
Hierarchy Digests	`hier`	**Cell Hierarchy Digests**	List of Cell Hierarchy Digests
Geometry Digests	`geom`	**Cell Geometry Digests**	List of Cell Geometry Digests
Libraries List	`libs`	**Libraries**	List libraries
OpenAccess Libs	`oalib`	**OpenAccess Libraries**	List OA libraries (with OA only)
Quit	`quit`	none	Exit *Xic*

1.4.4 Cell Menu

The **Cell Menu** contains command buttons to change the current cell, and to get information about cells in memory.

Cell Menu			
Label	**Name**	**Pop-up**	**Function**
Push	`push`	none	Edit subcell
Pop	`pop`	none	Edit parent cell
Symbol Tables	`stabs`	**Symbol Tables**	List of cell symbol tables
Cells List	`cells`	**Cells Listing**	List cells in memory
Show Tree	`tree`	**Cell Hierarchy Tree**	Display cell hierarchy

1.4.5 Edit Menu

The **Edit Menu** contains commands which provide panels for cell placement and property editing, and other features.

Edit Menu			
Label	**Name**	**Pop-up**	**Function**
Enable Editing	`cedit`	none	Enable/disable editing mode for current cell
Setup	`edset`	**Editing Setup**	Show **Editing Setup** panel
Create Cell	`crcel`	none	Create new cell
Create Via	`crvia`	none	Create a standard via
Flatten	`flatn`	**Flatten Hierarchy**	Flatten hierarchy
Join/Split	`join`	**Join or Split Objects**	Control join/split operations
Layer Expression	`lexpr`	**Evaluate Layer Expression**	Control layer expression evaluation
Properties	`prpty`	**Property Editor**	Edit properties
Cell Properties	`cprop`	**Cell Property Editor**	Edit cell properties

1.4.6 Modify Menu

The **Modify Menu** contains supplements the side menu with commands to undo/redo operations, and move, copy, and delete objects. Most of these commands have a faster keyboard equivalent.

Modify Menu			
Label	**Name**	**Pop-up**	**Function**
Undo	`undo`	none	Undo last operation
Redo	`redo`	none	Redo last undo
Delete	`delet`	none	Delete objects
Erase Under	`eundr`	none	Erase under objects
Move	`move`	none	Move objects
Copy	`copy`	none	Copy objects
Stretch	`strch`	none	Stretch objects
Chg Layer	`chlyr`	none	Move object to new layer
Set Layer Chg Mode	`mclcg`	**Layer Change Mode**	Set layer change mode for move/copy

1.4.7 View Menu

The **View Menu** contains commands which affect the presentation of the current design, including the selection of physical and electrical (schematic) modes.

View Menu			
Label	**Name**	**Pop-up**	**Function**
View	`view`	none	Set view in window
Physical or Electrical	`phys` or `sced`	none	Switch mode
Expand	`expnd`	**Expand**	Show detail in window
Zoom	`zoom`	dialog	Change window scale
Viewport	`vport`	sub-window	New drawing window
Peek	`peek`	none	Show layers in area
Cross Section	`csect`	sub-window	Show layers in cross-section
Rulers	`ruler`	none	Add transient gradations
Info	`info`	**Info**	Show cell/object parameters
Allocation	`alloc`	**Memory Monitor**	Show memory statistics

1.4.8 Attributes Menu

The **Attributes Menu** provides commands which affect the presentation of the design, such as the colors used.

Attributes Menu			
Label	**Name**	**Pop-up**	**Function**
Save Tech	`updat`	none	Save technology file
Key Map	`keymp`	none	Create keyboard mapping file
Define Macro	`macro`	none	Define a keyboard macro
Main Window		Attributes sub-menu	Set main window attributes
Set Attributes	`attr`	**Window Attributes**	Set rendering attributes for main window
Connection Dots	`dots`	**Connection Points**	Show connection dots in schematics
Set Font	`font`	**Font Selection**	Set text fonts used
Set Color	`color`	**Color Selection**	Set layer and other colors
Set Fill	`fill`	**Fill Pattern Editor**	Set layer fill patterns
Edit Layers	`edlyr`	**Layer Editor**	Add or remove layers
Edit Tech Params	`lpedt`	**Tech Parameter Editor**	Edit technology parameters

1.4.9 Convert Menu

The **Convert Menu** provides commands for importing and exporting designs to various non-native file formats, and for converting between file formats.

Convert Menu			
Label	**Name**	**Pop-up**	**Function**
Export Cell Data	`exprt`	**Export Control**	Create a cell data file
Import Cell Data	`imprt`	**Import Control**	Read a cell data file
Format Conversion	`convt`	**Format Conversion**	Direct file-to-file format conversions
Assemble Layout	`assem`	**Layout File Merge Tool**	Merge layout data
Compare Layouts	`diff`	**Compare Layouts**	Find differences between layouts
Cut and Export	`cut`	**Export Control**	Write out part of a layout
Text Editor	`txted`	**Text Editor**	Text edit cell file

1.4.10 DRC Menu

The **DRC Menu** contains commands associated with design rule checking.

DRC Menu			
Label	**Name**	**Pop-up**	**Function**
Setup	`limit`	**DRC Parameter Setup**	Set limits and other parameters
Set Skip Flags	`sflag`	none	Set skip flags
Enable Interactive	`intr`	none	Set interactive DRC
No Pop Up Errors	`nopop`	none	No interactive errors list
Batch Check	`check`	**DRC Run Control**	Initiate DRC run
Check In Region	`point`	none	Test rules in region
Clear Errors	`clear`	none	Erase error indicators
Query Errors	`query`	none	Print error messages
Dump Error File	`erdmp`	none	Dump errors to file
Update Highlighting	`erupd`	none	Update highlighting from file
Show Errors	`next`	sub-window	Sequentially display errors from file
Create Layer	`erlyr`	none	Write highlight error regions to objects on layer
Edit Rules	`dredt`	**Design Rule Editor**	Edit rules for layers

1.4.11 Extract Menu

The **Extract Menu** provides commands associated with the extraction of electrical information and netlists from the physical layout, and layout versus schematic checking.

Extract Menu			
Label	**Name**	**Pop-up**	**Function**
Setup	`excfg`	**Extraction Setup**	Set up and control extraction
Net Selections	`exsel`	**Path Selection Control**	Select groups, nodes, paths
Device Selections	`dvsel`	**Show/Select Devices**	Select and highlight devices
Source SPICE	`sourc`	**Source SPICE File**	Update from SPICE file
Source Physical	`exset`	**Source Physical**	Update electrical from physical
Dump Phys Netlist	`pnet`	**Dump Phys Netlist**	Save physical netlist
Dump Elec Netlist	`enet`	**Dump Elec Netlist**	Save electrical netlist
Dump LVS	`lvs`	**Dump LVS**	Save physical/electrical comparison
Extract C	`exc`	**Cap Extraction**	Extract capacitance using Fast[er]Cap
Extract LR	`exlr`	**LR Extraction**	Extract L/R using FastHenry

1.4.12 User Menu

The **User Menu** contains the script debugger, and the buttons that correspond to user-generated scripts.

User Menu			
Label	**Name**	**Pop-up**	**Function**
Debugger	`debug`	**Script Debugger**	Debug scripts
Rehash	`hash`	none	Rebuild **User Menu**
others	—	—	User scripts and menus

1.5 Database Overview

The core of *Xic* is the main database, which stores objects in a format that can be rapidly accessed spatially. The database, when given a rectangular region, will efficiently provide a list of contained objects whose bounding boxes overlap the given region. For example, when the user clicks or drags in a drawing window, the main database will quickly provide a list of the objects which overlap this area, so they may be shown as selected.

Each cell in memory has a database for each layer used by objects in the cell, plus a database corresponding to a dummy layer which contains the locations of subcell instances. The cells themselves are saved in one or more hash tables, the "symbol tables". The symbol tables allow cell data to be rapidly found by name. Cell name strings are saved in a common string table, so that address values can be used for efficient string comparison.

Each symbol table represents a self-contained design space, which can be rapidly switched between. *Xic* allows the user to define any number of symbol tables. Cells of the same name can not be saved in the same symbol table, but can exist in different symbol tables. Thus, for example, different versions of the same cell hierarchy can be kept in memory simultaneously, but the user can only view/edit using one symbol table at a time. This capability is used transparently by the geometry comparison functions, for example, in comparing two versions of the same cell.

The main database is organized as a tree, though the details are proprietary. This structure is self-balancing, unlike KD trees, thus there is no need to "rebuild" the database when objects are added or removed. The structure is optimized for rapid access, at a cost of time to build the structure. It is also optimized for low memory consumption, at a slight cost in speed.

When a file in loaded into the *Xic* "main" database, cell structures are created for each cell defined in the file. The cell structures contain trees for each layer used plus one for subcells if any, and are linked into the current symbol table.

The main database, with spatial access features, is not particularly efficient with regard to memory use. Large designs may not fit into available memory, depending on the machine. The physical memory limitation of the computer determines the maximum size of a file that can be read into *Xic* efficiently. Very roughly, the memory available should equal the size of the (uncompressed) GDSII file. If the file requires too much memory, *Xic* performance can become very sluggish due to page swapping, or in some cases the operating system will halt the process if memory limits are exceeded.

Although the design must reside in the main database for efficient cell editing, there are operations where this is not needed. There are provisions for handling extremely large files which can not be normally loaded.

1.5.1 Cell Hierarchy Digest

The Cell Hierarchy Digest (CHD) is a data structure designed to solve this problem. A CHD is an in-memory database which contains information about a hierarchy of cells, in a very compact manner. It holds no information about the geometry contained in the cells, but does contain offsets into the original layout file, so that through the CHD, the cell contents can be obtained reasonably quickly. Since the CHD uses a small fraction of the memory of the full design in the main database, it allows operations to be performed on very large designs with a modest computer.

The operations that can be performed with a CHD generally involve translation of a layout file into another layout file. For example, cell sub-hierarchies can be extracted, scaled, layers filtered or aliased, or cell names globally changed or aliased. The hierarchy can be flattened, filtered through a rectangular window and possibly clipped to the window, and empty cells (possibly produced by layer filtering) can be removed.

The CHD can also be used to view but not (directly) edit a large file. This is not as fast as viewing through the main database, but it is possible to view much larger files with a CHD.

There are also some novel ways to use CHDs in *Xic* to perform some limited editing. Reference cells in the main database are dummy cells that contain no data, but reference a cell hierarchy through a CHD. These cells can be instantiated in other cells normally. However, when written to a layout file on disk, they are replaced in output with the full referenced hierarchy obtained through the CHD. Thus one can use reference cells to assemble the top-level cell of a very large design. Each reference cell points to a sub-part of the design, kept in a separate layout file. When the top-level cell is written to disk, all of the parts will be extracted and combined into this file.

There is a cell override table which contains the names of cells in main memory. When enabled, when reading cell data through a CHD, cells in the override table will supersede cells in the original layout file. Thus, the cell override table provides a substitution mechanism. To perform minor editing in a hierarchy too large for main memory, one can

1. extract only the cells to be edited into main memory through a CHD,
2. edit these cells, and place their names in the override table, then
3. write a new layout file using the CHD, which will contain the new versions of the cells.

There is a related Cell Geometry Digest (CGD) which contains highly compact geometry collections on a per-cell/per-layer basis. A CGD can be linked to a CHD, with the total memory used still far smaller (by approximately a factor of 10) than the same cell hierarchy in the main database. With a linked CGD, when reading cell data through the CHD, the data are extracted from the CGD, avoiding accessing the original file on disk. This is usually faster.

1.5.2 Database Resolution

By default, *Xic* uses an internal resolution of 1000 units per micron. In releases prior to 3.0.12, this was internally hard-coded. As the dimensions used in integrated circuits continue to shrink, an option for higher resolution has been added.

The resolution can be set with the DatabaseResolution variable, which can be set to "1000", "2000", "5000", or "10000". If unset, 1000 units is used. This resolution applies only to physical data, electrical resolution is fixed at 1000.

This variable can be set only from the `.xicinit` file, which is read before the technology file, or the technology file. It can not be set or unset in a `.xicstart` file (read after the technology file) unless no technology file is read, or by any other means. It is important that the resolution be set before reading such things as DRC rules, since the rules contain resolution-dependent numbers which would be incorrect after a resolution change.

Superficially, changing the internal resolution has only subtle effects from the user's vantage point. Some of these are:

1. If not 1000, four digits following the decimal point are used when printing coordinates in microns, in many places in *Xic*. Otherwise, only three digits are used.
2. The ultimate zoom-in and grid spacing sizes are smaller for higher resolutions.
3. The size of "infinity", the maximum accessible size for the design, becomes smaller as resolution is increased, since coordinates are stored internally as 32-bit integers. For 1000 units, the field width is approximately 2 meters, which decreases to 20 centimeters at 10000 units. This should still be plenty for most purposes.
4. Layout files produced by *Xic* will use the internal resolution, so that no accuracy is lost.

Unless there is a specific need, it is recommended that users employ the default resolution.

Chapter 2

Xic Configuration and Startup

2.1 Graphics Support

Starting with Generation 4, *Xic* and all other Whiteley Research products used the GTK-2 graphical user interface toolkit exclusively. This replaced the Win32 graphical interface previously used under Windows and the GTK-1 interface used for other systems. Thus, all releases had precisely the same graphical interface, which greatly simplified documentation, maintenance, and development.

Presently (April 2024) GTK-2 is still the reference toolkit, but it is being replaced by Qt, another popular toolkit. The user interface is very similar, most of the changes are "under the hood". The tremendous amount of work to bring in a different toolkit is about complete, and Qt releases are in "beta" testing.

The reasons for the change are as follows.

- Starting with GTK-3, a new drawing layer named Cairo was introduced, and the previous drawing layer, which was a vaneer over the X-windows system, was eliminated. The problem was that the new drawing layer was not at all compatible with *Xic*, or any CAD type of tool requiring rendering precision and high performance. Cairo is geared for PowerPoint-type applications.

- GTK-2 is long-obsolete, and it will likely disappear at some point soon. The current GTK is GTK-4, which is not supported by *Xic* and probably never will be. *Xic* can build with GTK-3, but the result has serious flaws and shortcomings, and further development with GTK has stopped.

- GTK is a C library and *Xic* and *WRspice* are C++ programs, whereas Qt is also C++. The internal organization of the Qt version of the programs is far nicer and more concise and will be esier to maintain and extend.

- Qt has very good compatibility with Windows and macOS. There is no need to run an X server like XQuartz with the macOS Qt versions.

- Qt is contemporary software under active development, GTK-2 is very old.

2.2 Apple macOS Notes

In macOS, lurking beneath the pretty graphics is a complete Unix operating system based on FreeBSD, including support for X-windows. Most of the open-source software developed for Linux/FreeBSD has been ported to macOS, so for the Unix fans (like me), the operating system can look like Unix with a great graphical interface that also runs Macintosh applications. Overall, macOS is a very impressive desktop/laptop operating system.

The programs built with the GTK graphics toolkit require that an X-windows server be running, as X is used for graphics. This is not installed automatically. The Apple-sponsored X-server is called "XQuartz" and is available for download from the project web site (google "XQuartz download").

The programs built with the Qt graphics toolkit are native macOS applications and make use of direct access to the s creen, so there is no neew for XQuartz in this case.

Operation and behavior should be identical or very similar to the Unix/Linux versions of the programs.

2.2.1 Package Installation

Packages (precompiled program binaries) are being phased out in favor of building from source code. If you have compatible package files, they can be installed as described here.

The package distribution files are "flat" Apple package files. The installation procedure is pretty much the same as under Unix/Linux, using the `wr_install` script from a terminal window.

1. Download the necessary distribution files and scripts. Probably, the easiest course is to download everything in the distribution directory for your operating system, into a new, empty temporary directory. Presently, there are eight package files and two scripts. The installation method below uses the terminal window. When downloading completes, cd to the new directory.

2. Make sure that the `wr_install` and `wr_uninstall` scripts are executable (they might not be after the download). If not, use
   `chmod 0755 wr_install wr_uninstall`

3. The installation location is under `/usr/local/xictools`, and can not be changed. You will need to become 'root' in order to install packages. For future use of the automated updating features of *Xic* and *WRspice*, you will need to have your account enabled for the "sudo" command, i.e., your account name should be listed in "`/etc/sudoers`". If this is set up, the installer will ask for your password to enable installation. Otherwise you will have to use the su command to become root (which may not be enabled).

4. Run `wr_install` with the argument(s) being the names of the distribution files, or "`all`".

   ```
   ./wr_install all
   ```

   The argument "`all`" is equivalent to vt *.pkg. This automates the installation procedure. Do NOT change the names of the files, or `wr_install` will probably fail.

5. Add the `/usr/local/xictools/bin` directory to your shell search path, if you haven't done so. Once *Xic* and *WRspice* have been installed, updating could not be simpler. See the program documentation: In *WRspice*, give the commands "help passwd" and "help wrupdate". In *Xic* , with the mouse pointer in the main drawing window, type "?!passwd" and "?!update". These help

topics explain how to set up and use the update features. The programs will check for updates when started. The user can download and install updates when available. (Warning, packlages are deprecated and the update feature is not supported presently).

6. For a first-time installation, you will need to install and set up MacPorts. This provides the graphical libraries used by the programs. Following the instructions, install MacPorts for your operating system. Be sure to install the default package which installs in the default location (i.e., don't build from source and change the location). Note that the procedure also requires installing Apple xcode from the App Store. Once the basic installation is done, run, as root, the following commands:

```
port -v selfupdate
port install gtk2
port install gsl
port install python2_select
port install python27
port install autoconf
port install pkgconfig
```

7. Finally, if you are installing GTK versions and if you don't already have an X-server, you should install X-Quartz. XQuartz is the "official" X server for masOS and is recommended.

8. You're done, and should be able to run the programs. Furthermore, you should be able to build the *XicTools* from source, the MacPorts and xcode provide the necessary libraries.

2.2.2 Installation from Source

Presently, this is the recommended procedure for installing *Xic* and all of the *XicTools* on your system. See the README file at the top directory of the *XicTools* source tree from `github.com/wrcad/xictools`.

2.2.3 Un-Installation

To uninstall, the `wr_uninstall` script can be used. This takes care of file removal and updating the system package database.

2.2.4 Running the Applications

It is assumed that the user will be initiating the applications from a terminal window, as under Unix/Linux. The directory containing the program binaries (`/usr/local/xictools/bin`) should be in the shell search path. Then, all executables will be found by name. Operation is the same as under Unix/Linux.

2.2.5 MacBook Keyboard Mapping Issues

The Darwin64 releases work great on a MacBook Pro, however there are some keyboard mapping issues. Keys which have normal significance to *Xic*, such as **Home**, **Page Up** and **Page Down**, and the numeric keypad plus and minus keys are nowhere to be found. Yet, all functionality is present, but maps to alternative key combinations. One can run the **Key Map** command in the **Attributes Menu** if another mapping is needed.

The table below describes the default mapping.

MacBook Pro	Normal keyboard
Delete	**Backspace** The key labeled "**Delete**" actually sends a backspace character.
fn-Delete	**Delete** Press the **fn** key with **Delete** to get a real delete character.
fn-Left	**Home** Press the **fn** key with the left arrow for the center-full-view function in *Xic*.
fn-Up	**Page Up** Press the **fn** key with up arrow to get a page up code, used in *Xic* for displaying DRC errors.
fn-Down	**Page Down** Press the **fn** key with down arrow to get a page down code, used in *Xic* for displaying DRC errors.
These mappings are set in Apple releases only.	
fn-Enter	**KeyPad Enter** Press the **fn** key with **Enter** to get the numeric keypad enter code. This is mapped to the zoom-in action, as for **Numeric Plus**.
fn-Right	**End** Press the **fn** key with the right arrow to get the end key code. This is mapped to the zoom-out action, as for **Numeric Minus**.

Note that if you use a "normal" keyboard with your Apple computer, the two new mappings will be in addition to the normmal mappings.

The "secondary press" mentioned in Apple documents is button 3 (the right mouse button on a three-button mouse). You should probably change the track-pad settings in the **Preferences** to enable this. There is no button 2 (middle mouse button). You'll have to live without it, or get a three-button pointing device.

2.2.6 The Alt Key Issue

The MacBook Pro and probably other Mac machines lack a compatible **Alt** key. This **Alt** key is used in *Xic* as a menu accelerator, and for a button-press modifier. It is reasonable to live without it, but there is a fix.

The following fix works on my MacBook Pro. Create a file in your home directory named "`.Xmodmap`" containing the following two lines:

```
keycode 66 = Alt_L
add mod1 = Alt_L
```

This will map the left "option" button to **Alt**. The right option button will still do the normal Mac thing, i.e., send alternate character keycodes.

However, this depends on the left option key returning scan code 66, which may not be true on different hardware. The `xev` program can be used to find the actual scan code.

This will apply to all X applications, and the mapping will be recorded when the X server starts. You can also give the command

```
xmodmap -e "keycode 66 = Alt_L" -e "add mod1 = Alt_L"
```

which will re-map the keys for the current X session.

2.3 Microsoft Windows Notes

This section contains notes relevant to the Microsoft Windows release of the *XicTools*.

2.3.1 Installation and Setup

Packages (precompiled binary programs) are deprecated in favor of building the programs from source. If you have access to packages, they can be installed per the instructions in this section.

The packages come in self-extracting `.exe` files. Simply run the files to do the installation. The programs can later be uninstalled, either with the Windows software manager or by clicking the **Uninstall** icons in the **XicTools** program group in the **Start** menu.

The same process can be used to install updated releases – it is not necessary to uninstall first. A more convenient way to keep current is to use the updating feature of the help system (see 6.1.1). (Package updating is not currently available).

WARNING
The programs use an entry in the Windows Registry to find their startup files, etc. This entry is created by the installer program. Thus

1. The correct way to move an existing installation to another location is to uninstall the program, and reinstall to the new location using the standard distribution file. If you just move the files to a new location, the Registry won't be updated and the program won't run correctly.

2. You can not simply copy files from another machine when creating a new installation. The files must be installed through the distribution files, or the Registry entry won't be set.

The Registry entry used (by the `inno` installer program) is (for example)

```
HKLM\Software\Microsoft\Windows\CurrentVersion\Uninstall\Xic-4is1
```

The Windows `regedit` utility can be used to repair the Registry if necessary. The Windows `reg` utility can also be used to query and modify the Registry from the command line.

The programs are installed by default under `C:\usr\local`, which can be specified to the program installer. The installation directories will be created if necessary. All of the programs will install under a directory named "`xictools`" under the prefix, (for example, *Xic* files would be installed by default in `C:\usr\local\xictools\xic`). The structure of the tree is exactly that as under Linux, which simplifies compatibility. It is recommended that the default installation location be used, if possible.

Installing MSYS2

Starting January 2021, all Windows binaries are 64-bit, compiled under the MINGW64 subsystem of MSYS2 (64-bit version).

Although it may be possible to run the programs from other MINGW64 installations, for example Cygwin, it is recommended MSYS2 be installed. This replaces the `gtk_bundle` package that was previously used to supply graphical interface dlls.

When installing MSYS2 and *XicTools* programs, you are strongly advised to install in the default locations if possible.

Go to `www.msys2.org`. Download and run the installer, and follow the directions. It is easy. When done, you will have a shell window on the screen. You should type the following commands to load some things you will need.

```
pacman -S vim
pacman -S winpty
pacman -S mingw-w64-x86_64-gtk2
pacman -S mingw-w64-x86_64-gsl
```

This provides the vim editor, the winpty program described below, the GTK-2 libraries and dependencies, and the Gnu Scientific Library. These satisfy all references needed to run the *XicTools* programs.

Then, use the vim editor to edit the `.bash_profile` file that exists in the current (your home) directory. Add the following lines to the bottom of the file:

```
mount c:/usr/local/xictools /usr/local/xictools
export PATH="/usr/local/xictools/bin:$PATH"
export XT_HOMEDIR=c:/msys64/home/your_username
```

To run the programs in the future:
Click MSYS2 64bit in the **Start** menu, then MSYS2 MinGW x64 in the sub-menu. This brings up a shell window with the MinGW 64-bit support enabled. Note that there are multiple choices in the sub-menu so be sure to select the one specified, others may not work with the *XicTools* programs. Take some time to learn about MSYS2 and MINGW64, particularly the pacman package manager. You can install lots of useful software that runs in this environment.

The mintty Terminal Emulator Window

The terminal window used in MSYS2 is “mintty”. This does NOT work directly with native Windows programs that work with a console window, which includes the *XicTools* programs. One must run *WRspice* and possibly *Xic* with the “winpty” program. Just type winpty followed by the name and arguments of the program you want to run.

For example, the command to run *WRspice* is

```
winpty wrspice ...
```

where the ... represents the command line arguments to *WRspice*. The winpty program performs some magic that enables native Windows programs expecting to find a cmd.com type of window to work with the mintty.

subsubsectionMSYS2 Basics

The environment is a bash shell and environment very similar to Linux.

You bring up a new shell window from the MSYS2 64bit / MSYS2 MinGW x64 entry of the **Start** menu. There are multiple selections for MSYS2 in the **Start** menu, including MSYS2 MinGW x86 which is the MINGW 32-bit subsystem, MSYS2 MinGW x64 is the MINGW 64-bit subsystem used for *XicTools*, and MSYS2 MSYS is the MSYS subsystem without MinGW support. MinGW is "minimal Gnu (for) Windows" and is a library that provides a limited Linux-like programming environment under Windows. The *XicTools* programs use this environment and are thus Windows-native programs.

MSYS2 uses an underlying dll to provide a more complete Linux-like environment, similar to Cygwin. Installed programs are available for all three subsystems, though the search order differs. The *XicTools* do not use the dll, but the MSYS2 environment is very convenient for use with the tools. The MSYS subsystem provides the shell and other Linux-emulation commands. The MinGW subsystems provide libraries which enable porting of Linux programs to native Windows.

From the shell window, the file system installed under `c:\msys64` (or wherever the msys64 directory is installed) is taken as root. One can access the files and directories explicitly with a path starting with `/c` where the `c` can be any existing drive letter. Your home directory is `/home/`*yourname*, or equivalently `/c/msys64/home/`*yourname*.

It is important to realize that within MINGW and other native Windows programs including *XicTools*, the paths used by the programs are expected to be Windows paths, though they may use forward slashes. For example, in the shell `/usr/local` is rooted to the `msys64` directory, but within *Xic* `/usr/local` is rooted to the current drive, e.g., `c:/usr/local`.

In MSYS2, the mount command can provide an alias between the file system as viewed from the shell and that of Windows. The command

```
mount c:/usr/local/xictools /usr/local/xictools
```

will map the Windows directory `c:/usr/local/xictools` to the location `/usr/local/xictools` in the shell. From a shell window, `/usr/local/xictools` is equivalent to `/c/usr/local/xictools`, or in Windows `c:/usr/local/xictools`, the default *XicTools* installation location.

By default, the actual binary executable is run from a script (`.bat`) file, which is installed in the same directory as the executables, which by default is `C:\usr\local\xictools\bin`.

A program group named `XicTools` is created in the **Start** menu (or equivalent), from which the programs can be started. The programs can also be started from a command line, in either a Windows **Command Prompt** window or a MSYS2/Cygwin shell window. One will need to type the full path to the bat file (e.g., type "`C:\usr\local\xictools\bin\xic`" to start *Xic*). There are two ways to avoid having to type the whole path:

1. Add the directory to your search path. This is the `PATH` variable in the environment. This can be set in your **Command Prompt** window by giving a command like

   ```
   PATH=
   ```

   or the `PATH` can be set from the **Control Panel** (the procedure is described below for Windows 8).

2. The `bat` files can be copied from the installation location into a directory that is already in the search path, or to the current directory.

2.3.2 General Notes

The *XicTools* for Windows are supported on Windows 10 and later. The programs retain the "look and feel" of the Unix/Linux versions as much as possible, given the constraints of the Windows operating system.

Starting with Generation 4, the programs use the GTK-2 graphical interface toolkit, as used by the other releases. The native Win32 interface is gone. The GTK-2 libraries are supplied in a separate installation module. Installation is mandatory, but the libraries are quite static so will not require much attention after the initial installation.

Most basic features are available under Windows. Some of the more advanced features are not.

- There is presently no support for the Tcl/Tk or Python script language plug-ins. There is also no support for the OpenAccess plug-in.
- There is no provision for remote running of the programs as with the X window system in Linux.
- Under Unix/Linux, when the program crashes (of course, a very rare occurrence!), the `gdb` debugger is called to generate a stack trace, which is emailed to Whiteley Research for analysis. Since it is rare to find `gdb` on a Windows system, an alternative is built in. This produces a file named *progname*.`stackdump`, which is emailed (if possible) to Whiteley Research.
- Windows does not provide a reliable interface for internet mail, so the email clients and crash-dump report in the *XicTools* may not work. The mail in *XicTools* works by passing the message to a Windows interface called "MAPI", which in turn relies on another installed program to actually send the mail.

  To get this working in Windows 8, I had to download and install something called "live mail" from Microsoft, which eventually worked. This app supports MAPI, apparently the Windows 8 Mail app does not(?). The Windows 8 app also does not work with POP3 servers, solidifying my disrespect.

The "environment variables" mentioned in the *Xic*/*WRspice* documentation are available, and can be set in a **Command Prompt** window with the "`set`" command before starting the programs, or from the **System** entry in the **Control Panel** (or wherever this capability lives in your version of Windows). Only the latter method works if the programs are started from an icon or menu.

Directory path names used by the programs can use either '/' or '\' as the directory separator character, interchangeably. The path can also contain a drive specifier.

The path variables used by *Xic* that contain lists of directory paths must use either a space or ';' (semicolon) as a separator. Under Unix, the separation characters are space and ':' (colon).

The text files used by the programs can have either DOS or Unix line termination. Text files produced by the programs under Windows will use the DOS format.

Under Windows, where the concept of a "home directory" is somewhat tenuous, the programs will look for environment variables, particularly HOME, and if found interpret the value as a path to the home directory. This is true when programs look for startup files. When the program is started from an icon or shortcut, and the start directory is not explicitly set in the icon properties (it defaults to `C:/`), the current directory will be the home directory, rather than `C:/`.

Those used to a Unix environment are encouraged to download and install the Cygwin tools. These include most of your favorite Unix commands, plus a complete compiler toolchain for application development. In particular, the bash shell is quite useful, as it provides a "DOS box" that responds to Unix shell commands, and from which one can execute shell scripts. The tools can be downloaded as individual modules.

If it is needed and does not exist, *Xic* and *WRspice* will create a `\tmp` directory on the current drive. This will contain temporary files, used by the programs. These should be removed automatically when the programs terminate, but if not the files can be safely deleted if *Xic* and *WRspice* are not running.

2.3.3 Setting Environment Variables

By running Cygwin, the setting of environment variables and similar becomes very familiar to a Linux user. In particular, running *Xic* from a Cygwin bash-shell window emulates pretty well the Linux experience. This is a recommended approach for those familiar with Unix/Linux.

Otherwise, environment variables can be set manually in a **Command Prompt** window from which the programs are run. The `bat` files can be modified and "`set`" lines added, as an option to avoid manual setting of variables that should always be in force. Another option is to set the system default environment variables. Be aware that all other programs will see the variables. Setting the system environment variables is probably something to avoid if possible. If you insist, here is the procedure for Windows 8.1. Other supported Windows releases are probably not horribly different.

1. Go to the infamous **Start** page, click on the circle with down arrow icon near the bottom-left corner. This shifts to the **Apps** page.
2. Find the **Control Panel**, it is listed on the **Apps** page under **Windwos System** You can use the search tool if necessary. Eventually, you'll find the icon, then click it to bring up the **Control Panel**.
3. Click **System and Security**. The display will change to a new set of choices.
4. Click **System**.
5. Click **Advanced system settings** along the left. This brings up a **System Properties** window.
6. Click the **Environment Variables** button near the bottom of the **Advanced** page (this page should be shown initially).

There are a couple of things one may want to do here, as examples.

1. Add the *XicTools* `bin` directory to the system search path.

   Scroll the lower **System variables** window to find the `Path` entry. Click on this to select it. Click the **Edit...** button below, which brings up a text entry window. In the **Variable value** window, scroll all the way to the right, and add, for example (use the actual paths if different on your system)

   ```
   ;c:\usr\local\xictools\bin
   ```

Check the spelling, and make sure there is no white space, and that the character before the '`c`' is a semicolon, and the character that follows the '`c`' is a colon. Then click the **OK** button.

2. Add a `HOME` variable to define a "home directory".

   Press the **New...** button below the UPPER listing window (not the one you just used). This brings up a text entry as we saw before. Enter `HOME` for **Variable name**, For **Variable value**, enter a path to some directory which you want to be your "working" directory, where *Xic* and *WRspice* will look for startup files, etc. Enter the full path to this directory. Check spelling, Click **OK**.

Finally, click the **OK** button at the bottom of the window, we're done.

When a program is started from an icon, an icon property specifies the directory where the program logically starts from. This is the apparent current directory seen by the user when running the program. By default, this is usually sonething like "`C:\`", which is not a good choice. The user should have a directory dedicated for this, and the following procedure can be used to cause the programs started from an icon to start in this directory.

1. Go to the **Start** page, click on the circle with down arrow icon near the bottom-left corner. This shifts to the **Apps** page.
2. Find the **XicTools** program group. There should be entries for the programs that you have installed.
3. For each program:

   (a) Click on the program icon with the RIGHT mouse button. An icon banner along the bottom of the screen will appear.

   (b) Click on **Open File Location**. This brings up a listing showing the **XicTools** programs.

   (c) Above the list, find the **Properties** icon and click it. This brings up a multi-page **Properties** pop-up.

   (d) In the **Shortcut** page, change the entry in the **Start in** entry area to a full directory path to the directory where the program should start. This might be the same directory that you used for the `HOME` environment variable.

   (e) Then click the **Apply** button, and click **OK** if there is a confirmation pop-up.

This applies to the icon in the **Apps** page. Other icons can be set similarly.

2.4 Command Line Options

The following syntax applies when *Xic* is invoked from the command line. Arguments not recognized as options are expected to be files containing layout information in supported formats. The first such file (if any) will be loaded into the editor. Subsequent files can be loaded sequentially with the **Open** command.

```
xic [-F filetool_args] | [ [-Bbatch_opt | -S[port] [-C | -C1] [-E]
[-Ggeometry_spec] [-Hdirectory_path]   [-Kpassword] [-Lserverhost[:port]]
[-Rprefix_path] [-T[extension]] [toolkit_options]   [filename ...]  ]
```

Xic will accept command line options common to applications designed around the GTK user interface toolkit. In addition, there are a few command line options used exclusively by *Xic*. Options are keyed by a hyphen '-', and can not be grouped. Above, the square brackets indicate that the specification is optional (which applies to all arguments), and the '|' symbol is a logical "OR" operator indicating that one may specify one of the surrounding forms.

-B*batch_opt*

Xic supports a batch mode of operation, where *Xic* will run a script or perform certain commands without graphics. The form for this option is one of

-B*scriptfile*[,*args*...]
-B-*command*[@*arguments*]

Batch mode will be described in 4.4.

The -C and -C1 options apply only to "pseudo-color" displays. These are displays with "8-bits" or "256 colors", found on older workstations. By default, *Xic* uses a large percentage of the system colormap. If there are insufficient colormap entries available, *Xic* will create its own virtual colormap, which is loaded when an *Xic* window has the keyboard focus. A problem is that some X terminals and emulators apparently do not support virtual colormaps, or do so improperly. Also, the use of a virtual colormap can be annoying. For these reasons, options have been provided to limit colormap usage, and avoid creation of a virtual colormap.

-C

This option applies only in pseudo-color visual modes. The -C option, if given, will prevent *Xic* from allocating private colors from the system colormap. Instead, it will use cells shared with other applications. The colormap usage can be dramatically reduced by this option. The cost is 1) the colors may not be quite "right" if the colormap is already heavily used by other applications, 2) there is no blinking, 3) the colors can not be changed, and 4) highlighting may be difficult to see, as for the -C1 option. A second copy of *Xic* running with the same technology file as the first will use no additional colormap space. A virtual colormap is never produced if the -C option is given. This option is recommended primarily for users who want to run multiple copies of *Xic* without the virtual colormap.

-C1

This option applies only in pseudo-color visual modes. The -C1 option similarly saves colormap space by directing *Xic* to allocate single-plane cells. By default, and if sufficient colormap space is available, *Xic* will allocate "dual-plane" color cells for the layer rendering colors. These cells contain two pixel values, one representing the color, and one which is white. The white pixel is addressed during highlighting, and having one white pixel per layer ensures that the exclusive-or drawing mode always produces white highlighting.

Single-plane color cells use half the colormap space of dual plane cells. However, the exclusive-or highlighting is only guaranteed to be white over the background, and the highlighting can take any color over the layers. This can sometimes be difficult to see.

-E

The -E option signals *Xic* to start in electrical mode. The default is to start in physical mode.

-F

This option must be the first given, and arguments that follow must be appropriate for the *FileTool* utility (see Appendix G). The program will behave as the command-line *FileTool* program, which can perform various manipulations and diagnostics on layout files.

If the `xic`, `xicii`, or `xiv` binary executable files (or Windows `.exe` equivalents) are copied or linked under the name "`filetool`" ("`filetool.exe`" under Windows), the new program will behave as a *FileTool* when invoked.

-G*geometry_spec*

The *geometry_spec* is an X-style window geometry specification, which allows the main window size and position to be specified. There is no space between `-G` and the specification. The command line specification will override the XIC_GEOMETRY variable. The format of the *geometry_spec* is described with the environment variable.

-H*directory_path*

Giving this option will cause *Xic* to start in *directory_path* as the current working directory. Note that there is no space between the "`-H`" and the directory path.

-K*password*

The password used to enable use of encrypted scripts can be given to *Xic* on the command line with this option. Note that there is no space between the "`-K`" and the password. As the password can contain almost any character, if the password contains characters which could be misinterpreted by the shell, the password should be quoted, e.g., `-K'`*password*`'`.

If no password is given to *Xic* with the `-K` option, a default password is effective. The default password has a key that is compiled into the executable file, which can be changed with the `wrsetpass` utility. The "factory" default password is

Default password: `qwerty`

The password set with the `-K` option overrides the default password. The password can also be set with the `SetKey` script function.

If the `.xicinit` or `.xicstart` file, or the function library file, or a script run from batch mode is encrypted, the encryption password must be given to *Xic* with the `-K` option, or be the default password. As the password can be changed with the `SetKey` script function, **User Menu** scripts can in principle use different passwords, which must be set before the script is executed.

-R*prefix_path*

If given, the *prefix_path* internally replaces "`/usr/local`" when *Xic* composes directory paths to search for startup files. This will override the value of the XT_PREFIX environment variable. This is one method of specifying to *Xic* the startup file location, if the distribution was installed in a non-default location. Under Windows, the installation location is saved in the registry and is available to *Xic*, so *Xic* should be able to find its startup files without this option.

-S[*port*]

If the -S option is given, *Xic* will run in server mode. In this mode, *Xic* runs in the background as a daemon process, serving requests through a communications port. This mode will be described in 4.5. The option can be immediately followed (no space) by a port number to use for connections.

-T[*extension*]

The -T*extension* option is used to designate a particular technology file, which is a file used by *Xic* to initialize itself to a particular manufacturing process and set of user preferences. The technology file has a name of the form `xic_tech` or `xic_tech.`*extension*, the base name is always "xic_tech", but there may be an arbitrary extension (characters other than '.' following '.'). If no -T option is given, then the `xic_tech` file is used. Otherwise, the *extension* given in the option will signal *Xic* to use the technology file with the same extension. Note that it is allowable to start *Xic* without any technology file, which is the effect of giving just the -T without any extension. Note that there must not be any space between the T and the extension.

The graphical interface accepts the following options. These options are not processed by *Xic*, but are intercepted by the graphics subsystem and affect the interface to the X-window system. The multiple forms are equivalent.

-d *dispname*
-display *dispname*
--display *dispname*

This option specifies the name of the X display to use. The *dispname* is in the form

[*host*]:*server*[.*screen*]

The *host* is the host name of the physical display, *server* specifies the display server number, and *screen* specifies the screen number. Either or both of the *host* and *screen* elements to the display specification can be omitted. If *host* is omitted, the local display is assumed. If *screen* is omitted, screen 0 is assumed (and the period is unnecessary). The colon and (display) *server* are necessary in all cases. If no display is specified on the command line, the display is set to the value of the environment variable DISPLAY.

-name *string*
--name *string*

This option provides an alternative name to the application, as known to the X window system. The application name is used by X to apply resource specifications.

--class *string*

This option provides an alternative class name to the application, as known to the X window system. The application class name is used by X to apply resource specifications.

-synchronous
--sync

This option indicates that requests to the X server should be sent synchronously, instead of asynchronously. Since the X system normally buffers requests to the server, errors do not necessarily get reported immediately after they occur. This option turns off the buffering so that the application can be debugged more easily. It should never be used with a working program.

--no-xshm

In releases running under the X-Window system (Unix/Linux), *Xic* will use the MIT-SHM shared memory extension if the X server supports this extension, and the server is running on the local machine. This allows image data to be transferred to the X server via shared memory, which is faster than the normal X socket interface. Screen updates may be faster as a result.

Giving the option --no-xshm on the command line will prevent use of this extension, if for some reason this is necessary.

--v

If this option is given, *Xic* will print a string containing three tokens and exit. The tokens are

version osname arch

for example "`4.3.11 LinuxCentos7 x86_64`".

--vv
If this option is given, *Xic* will print a CVS-style tag string and exit. The format is, for exmple, "`xic-4-3-1`".

--vb
If this option is given, *Xic* will print the build date and exit.

Any words found in the command line that are not recognized as options will be interpreted as files to load into *Xic* for editing. The files will be loaded in order of their appearance, with the first file loaded at startup, and the other files loaded in response to an **Open** command.

2.5 *Xic* Environment Variables

Environment variables are keyword/value pairs that are made available to an application by the command shell or operating system. The value of an environment variable is a text string, which may be empty. Environment variables can be set by the user to control various defaults in *Xic*.

2.5.1 Unix/Linux

Environment variables are maintained by the user's command shell. It is often convenient to set environment variables in a shell startup file such as `.cshrc` or `.login` for the C-shell or `.profile` for the Bourne shell. These files reside in the user's home directory. See the manual page for your shell for more information.

For the C-shell, the command that sets an environment variable is

`setenv` *variable_name* [*value*]

For example,

```
setenv XT_DUMMY "hello world!"
```

Note that if the value contains white space, it should be quoted. Note also that it is not necessary to have a value, in which case the variable acts as a boolean (set or not set).

In the C-shell, one can use `setenv` without arguments, or `printenv`, to list all of the environment variables currently set.

For a modern Bourne-type shell, such as `bash`, the corresponding command is

`export` *variable_name*[=*value*]

In this type of shell one can list the variables currently set by giving the `set` command with no arguments.

2.5.2 Microsoft Windows

Under Windows, environment variables can be set in a DOS box with the `set` command before starting the program from the command line, or in the `AUTOEXEC.BAT` file, or from the **System** entry in the **Control Panel**. Only the latter two methods work if the programs are started from an icon. If using a Cygwin bash-box, environment variables can be set in the startup file as under Unix.

2.5.3 *XicTools* Environment Variables

The following environment variables are used by all *XicTools* programs.

CYGWIN_BIN

This variable applies only when running under Microsoft Windows, and Cygwin is installed. Cygwin is Linux-like environment and tool set which is a very useful adjunct to Windows. In particular, it provides a bash shell with standard Linux commands, and an X server, among many installable features.

XicTools programs will in some cases, such as when popping up a shell window, look for a Cygwin program. If the Cygwin program binaries (`.exe` files) are located in `/bin` or `/cygwin/bin` on the current disk drive, they will be found automatically. Otherwise, this variable can be set to the Windows path, including a drive letter if necessary, to the directory containing the Cygwin binaries. This is not necessarily the path one perceives from within Cygwin, since the *XicTools* programs do not know about the Cygwin mount points or symbolic links. The path is the one that would be seen from a DOS box, with forward or reverse slash directory separators.

XT_PREFIX

All of the *XicTools* programs respond to the XT_PREFIX environment variable. When the tools are installed in a non-standard location, i.e., other than `/usr/local`, this can be set to the directory prefix which effectively replaces "`/usr/local`", and the programs will be able to access the installation library files without further directives. The *Xic* -R command line option can also be used for this purpose. This should not be needed under Windows, as the Registry provides the default paths.

XT_HOMEDIR

Under Windows, the user's "home" directory is determined by looking at environment variables.

In Linux, the HOME environment variable is set the the user's home directory, and this is also true under Windows if using a Linux emulation package such as Cygwin or MSYS. However, in this case HOME will be relative to the file system as seen within the emulator, and not the actual Windows file system as seen in *Xic* or *WRspice* which are Windows-native programs. Therefor, the HOME environment variable is ignored under Windows.

Instead, the programs will first look for XT_HOMEDIR. This should be set to the Windows path to the user's MSYS2 or Cygwin home directory. For example, this can be done from the `bash_profile` file by adding a line

`export XT_HOMEDIR=c:/msys64/home/`*yourlogin*

Setting this will allow *Xic* and *WRspice* to find files in the user's MSYS2 home directory, even though the programs are Windows native and don't know the MSYS2 paths.

The deprecated XIC_START_DIR variable is checked next, and if found its value is taken as the user's home directory in the same manner.

If not found, the HOMEDIR and HOMEPATH variables, if both are found, are concatenated to yield the home directory path. In the unlikely event that these are not set, the USERPROFILE variable is checked, and if all else fails, "`C:\`" is assumed. The HOMEDIR/HOMEPATH and USERPROFILE variables are set by Windows, at least in some Windows versions.

Under other operating systems, the home directory is well-defined and is obtained from operating system calls.

Under Windows, if *Xic* finds itself in the `C:\` directory on startup, it will change the working directory to the home directory. This is the default when starting from the Windows **Start Menu** or otherwise from an icon, unless the icon property is changed.

XTNETDEBUG

If the variable XTNETDEBUG is defined, *Xic* and *WRspice* will echo interprocess messages sent and received to the console. In server mode, *Xic* will not go into the background, but will remain in the foreground, printing status messages while servicing requests.

Linux and FreeBSD releases can use an included local memory allocation package. In earlier *Xic* releases, this allocator, rather than the allocator provided by the operating system, was used by default. In 32-bit releases, the local allocator was often able to allocate more memory than the allocators provided by the operating system. It also provided custom error reporting and statistics.

This feature is now disabled, as in modern operating systems there is dubious benefit, and it can produce stability problems in some cases. However, if this variable is set in the environment when *Xic* is started, the local allocator will be used. The interested user is encouraged to experiment.

XT_SYSTEM_MALLOC

This variable was once used to disable the internal local memory allocator, which in earlier releases was enabled by default. Currently, this variable is ignored.

XT_GUI_COMPACT

When set, no extra space is allowed around pushbutton contents in the graphical interface. Such space can cause side menu button images to be truncated on low-resolution displays if the theme in use imposes too much space. Setting this variable is a quick fix for this problem, though one could also change the theme.

2.5.4 *Xic* Environment Variables

The following paragraphs describe the environment variables which are relevant to *Xic*.

FORCE_XICII

If this variable is set when *Xic* starts, the program will run as *XicII*. *XicII* was a reduced feature set (layout editor only) version of *Xic* available at lower cost. Operating in this mode may simplify things for some users. One can create an "`xicii`" program with the following shell script:

```
#!  /bin/sh

FORCE_XICII=1 xic $*
```

FORCE_XIV

If this variable is set when *Xic* starts, the program will run as *Xiv*. *Xiv* was a reduced feature set (layout viewer only) version of *Xic* available at lower cost. Operating in this mode may simplify things for some users. One can create an "`xiv`" program with the following shell script:

```
#!  /bin/sh

FORCE_XIV=1 xic $*
```

XIC_HOME

This environment variable applies only to the *Xic* program. If found in the environment when *Xic* starts, it is expected to contain a path to the *Xic* installation area or equivalent, which defaults to "`/usr/local/xictools/xic`". This overrides XT_PREFIX if that environment variable is also found.

There is an important subtlety when using this variable. Although it allows *Xic* to find its startup files anywhere, only the directory structure implied by XT_PREFIX, that is, for *Xic*,

```
$XT_PREFIX/xictools/xic
```

is compatible with the program installation script. The variable is perhaps useful for pointing *Xic* toward a secondary set of startup files, perhaps heavily customized by the user, which may reside in an arbitrary location.

XIC_GEOMETRY

This can be set to an X-style geometry string, to specify the default size and position of the *Xic* main window.

If the geometry has been specified, *Xic* will use it to position and size the main window (if the window manager permits this). The geometry specification, used to define window size and position, is a string in the form

*width*x*height*+*xoff*+*yoff*

where *width*, *height*, *xoff*, and *yoff* are numbers representing screen pixels. The "x" or "X" between the *width* and *height* is literal. A plus sign '+' or minus sign '−' must appear ahead of *xoff* and *yoff*.

+*xoff*
The left edge of the window is to be placed *xoff* pixels in from the left edge of the screen.

−*xoff*
The right edge of the window is to be placed *xoff* pixels in from the right edge of the screen.

+*yoff*
The top edge of the window is to be *yoff* pixels below the top edge of the screen.

−*yoff*
The bottom edge of the window is to be *yoff* pixels above the bottom edge of the screen.

XIC_TECH_DIR

The value is a path to a directory. If given, the directory is searched for the technology file, if not found in the current directory, and before other locations are checked.

XIC_TMP_DIR, TMPDIR

By default, *Xic* uses the directory `/tmp` for temporary files. In some installations, this directory may be too small to accommodate the large files needed by *Xic*, for example when producing hard copy plots. An alternative directory for temporary files can be specified with the XIC_TMP_DIR environment variable (which has precedence) or with the TMPDIR variable, which is a Unix standard. One of these should be set to a path to a directory to use for temporary files, if necessary.

XIC_LOGDIR

The variable XIC_LOGDIR can be set to a path to a directory which will be used to store certain log files produced while *Xic* is running. The location used for the log files is the first defined of XIC_LOGDIR, XIC_TMP_DIR, TMPDIR, or `/tmp` if none of these variables is defined. The log files are removed on normal exit.

XIC_MENU_RIGHT

If the variable XIC_MENU_RIGHT is defined in the environment, *Xic* will place the side menu and layer table to the right of the main window. The default to to place the menu at the left.

XIC_HORIZ_BUTTONS

If this variable is set in the environment when *Xic* starts, the buttons in the side menu will be arrayed horizontally across the top of the main window instead.

XIC_PLUGIN_DBG
If this variable is set in the environment when *Xic* starts, error messages concerning plug-in loading will be printed in the console window. Without this set, *Xic* will simply silently not load a plug-in if an error occurs. These diagnostic messages can help identify why the plug-in is not being loaded, and are instrumental in tracking down problems when the user expects success.

This variable is deprecated. Under Windows, it is interpreted in the same manner as XT_HOMEDIR.

XIC_EXIT_CMD
If the environment variable XIC_EXIT_CMD is set to a command string, that command will be executed when *Xic* exits. If the command string contains spaces, the command should be quoted. For example, using

```
setenv XIC_EXIT_CMD "/usr/games/fortune -o"
```

may print a rude limerick on some installations. This feature may have less frivolous uses, however.

XIC_SYM_PATH, XIC_LIB_PATH, XIC_HLP_PATH, XIC_SCR_PATH
There are four additional environment variables used to specify locations where *Xic* is to look for certain types of files. These variables are XIC_SYM_PATH, XIC_LIB_PATH, XIC_HLP_PATH, and XIC_SCR_PATH. These variables are described in the next section.

The internal default values for the paths assume that the installation location is the standard place under `/usr/local`, or if the XT_PREFIX variable is set, that value will be taken instead of "`/usr/local`".

XIC_DOCS_DIR
The environment variable XIC_DOCS_DIR can be set to an alternate location for the archive of release notes. This location is searched in the **Release Notes** command in the **Help Menu**. The default location is `/usr/local/xictools/xic/docs`, or, if XT_PREFIX is set, its value will replace `/usr/local`.

XIC_OASO_PATH
Plugins are normally found in the `plugins` directory in the installation area, which by default is

```
/usr/local/xictools/xic/plugins
```

This variable can be set to the full path to the OpenAccess plug-in, which *Xic* will attempt to load on program startup instead of looking in the default location.

XIC_PYSO_PATH
Plugins are normally found in the `plugins` directory in the installation area, which by default is

```
/usr/local/xictools/xic/plugins
```

This variable can be set to the full path to the Python plug-in, which *Xic* will attempt to load on program startup instead of looking in the default location.

XIC_TCLSO_PATH
Plugins are normally found in the `plugins` directory in the installation area, which by default is

```
/usr/local/xictools/xic/plugins
```

This variable can be set to the full path to the TclTk or Tcl-only plug-in, which *Xic* will attempt to load on program startup instead of looking in the default location.

XIC_LIBRARY_PATH
: This applies to Linux and macOS only. If set, the value will be prepended to the `LD_LIBRARY_PATH` in the *Xic* wrapper script. This can be used to point to installed libraries needed for plugins, for example the OpenAccess libraries, without having to set `LD_LIBRARY_PATH` in the environment.

XIC_NO_MAC_MENU
: (Qt macOS only) By default, *Xic* will use the Apple menu in the upper left of the screen. This has no tooltips and can not be selected programmatically. It this variable is set, *Xic* will use Linux-style menus in the main window which behave as they do in Linux, enabling access to macros.

XICNOMAIL
: If the variable XICNOMAIL is set, no mail will be sent during a crash. If a fatal error is encountered, a file named "`gdbout`" is created in the current directory, which contains a stack backtrace from the stack frame of the error. Despite the name, the file is generated internally on all platforms, and no longer makes use of the `gdb` program.

By default, this file will be emailed to Whiteley Research for analysis. However, the emailing can be suppressed by setting this variable in the environment. The `gdbout` file is produced in any case, and would be very useful to Whiteley Research for fixing program bugs.

XTNOMAIL
: This has the same effect as XICNOMAIL but also prevents email from the *WRspice* program.

SPICE_HOST, SPICE_EXEC_DIR, SPICE_EXEC_NAME
: When connecting to SPICE in the **run** command, the SPICE_HOST variable is used to set the name of a remote SPICE host which provides SPICE service. The name can optionally be followed by a colon and a port number, if a non-default port is used by the SPICE server. The SPICE_EXEC_DIR environment variable provides the directory which contains the `wrspice` executable, which may need to be identified to *Xic* if it is other than `/usr/local/bin`. The SPICE_EXEC_NAME environment variable can be used to provide an alternate name for the `wrspice` executable, if it has been changed. The default is, of course, "`wrspice`". Each or these environment variables can be overridden by a corresponding internal variable, which can be set with the **!set** command.

IMSAVE_PATH
: The printing interface includes a driver for generating image files in various formats. A few formats are handled internally, however vastly more are available through other software that may be available on the system. The driver can usually locate these programs by looking in standard places, however, if the programs exist but can't be located, this variable can be set to a colon-separated list of directories to search for the executables. This applies to Unix/Linux/macOS only. See the description of the **Image** print driver in 8.6.2 for more information.

2.6 *Xic* Search Paths

There are four search paths used by *Xic*. Search paths are lists of directories, which are searched in left-to-right order for files of a particular type. In addition to search paths, *Xic* provides a "redirect file" mechanism for finding files, which supplements the search path. If a specific file is being sought, the first file with matching name is used. The format used for search path strings can be one of two forms:

Unix-shell style: `( directory`*1* `directory`*2* `...  directory`*N* `)`
: The tokens are separated by white space. If white space appears in a directory entry, that entry should be single or double quoted. The entire path should be enclosed in parentheses. Space between the parentheses and directory names is optional.

Examples:

```
( . )
( /usr/local/bin "/Program Files/xic/stuff" ~/work )
```

This format is the same in Windows and Unix releases, however in Windows, back and forward slashes are equivalent, and the drive specifier can appear in the entries.

Traditional search path: `directory`*1*`:directory`*2*`:...:directory`*N*
The entries are separated by a special character, which is a colon '`:`' in Unix/Linux, and a semicolon '`;`' in Windows. There should be no white space that is not part of a file/directory name. An entry should be single or double quoted if it contains the separation character. In the examples here, a colon is used, which in Windows must be converted to a semicolon. The separation character is optional at the front or end of the path, unless it is needed to delimit white space that is part of an entry.

Examples:

```
.
/usr/local/bin:/Program Files/xic/stuff:~/work
```

In earlier *Xic* releases, parsing was fairly loose, and in particular hybrids of the two formats would be accepted. This is not true in the present release, due to support for white space in path entries. The format used in a path string must be consistent.

The following special symbols are recognized in entries:

`.`	The current directory
`..`	The parent directory of the current directory
`~`	The user's home directory (Unix) or the content of the HOME environment variable (Windows)
`~joe`	The home directory of user `joe` (Unix only, no substitution in Windows)

The four paths are the design data path, the library path, the help path, and the script path. The design data path is used to locate design data files, consisting of native cell, archive, and library files. The library path is used to locate the technology file, device and model libraries, and various other configuration files. The help path contains files for the help system, and the script path contains executable scripts and libraries which appear as commands in the **User Menu**.

These paths can be set in the technology file, the `.xicinit` or `.xicstart` initialization files, or by use of environment variables, or with the **!set** command. A specification in the `.xicinit` will override specification in the environment, which is in turn superseded by a specification in the technology file, and the `.xicstart` file supersedes the technology file. Once *Xic* is running, the **!set** command can be used to set or examine the search paths. Similar commands exist in the script interpreter interface function library.

In addition, the design data path is augmented with any path preceding a native cell file to open in the **Open** command. By default, the path is added to the beginning of the present design data path. For example, suppose a design hierarchy exists in the directory `/usr/work`. If the user enters `/usr/work/maincell` in response to the prompt which appears after pressing the **Open** button, then the file maincell is opened for editing, and the directory `/usr/work` is added to the front of the design data path. Once the design data path is updated, the cells in that path can be accessed by their base file name only. The treatment of any path which is given with a native cell to open in the **Open** command can be altered with the NoReadExclusive and AddToBack variables.

The use of paths facilitates user customization of *Xic*, particularly when the directories used in the system installation are not writable by the user. By installing a different search path, the user can augment or substitute for the system default files and libraries.

Below are the environment variable names and internal defaults:

Design Data Path
- variable: Path
- environment: XIC_SYM_PATH
- default: `( . )`

Library Path
- variable: LibPath
- environment: XIC_LIB_PATH
- default: `( . /usr/local/xictools/xic/startup )`

HelpPath
- variable: HlpPath
- environment: XIC_HLP_PATH
- default: `( /usr/local/xictools/xic/help )`

ScriptPath
- variable: ScriptPath
- environment: XIC_SCR_PATH
- default: `( /usr/local/xictools/xic/scripts )`

If the XT_PREFIX environment variable is set, its value will be taken instead of "`/usr/local`" in the defaults.

The "variable" field in the table above provides the name of the variable, which can be altered with the **!set** command to set the path. Unlike other variables, these are always defined and cannot be unset. The same name is also used as a keyword in the technology file.

Files containing cell data, whether *Xic* native, GDSII, or some other format, are expected to be found in a directory along the design data search path. The first file found matching the name requested is opened. Normally, it is desirable to include the current directory '.' in the design data path, otherwise files located in the current directory will not be found.

The technology file, `device.lib` file, `model.lib` file and other model files are found along the library path.

The search behavior of the library path is slightly different from the other paths, in that an attempt is made to open a file in the current directory before looking through the search directories. Thus, the current directory '.' is always logically at the head of the library path. There is no problem if '.' is also explicitly defined in the path. A consequence is that startup files that exist in the current directory will *always* have precedence over files located in other directories.

Each directory in the help path is expected to contain help database files. These files use names with an extension "`.hlp`". The directories may also contain graphics files used by the help system. Changing this path allows the user to provide their own help files for the custom functions (scripts) which appear in the **User Menu**, for example, or to add information topics, such as about local design rules, to the database.

The scripts and related files are found along the script path. Only files which have the extension "`.scr`" are taken as scripts. The directories in this path may also contain script menus, with extension

"`.scm`", and files named "`library`" which contain subroutines used by other scripts. Whenever the script path is changed, a **rehash** is performed, i.e., the **User Menu** is rebuilt.

2.7 Redirect Files

Redirect files are an adjunct to the search path mechanism used by *Xic* for finding files. Redirect files are files created by the user, that tell *Xic* about additional locations to search for input files.

Redirect files **must** be named "`xt_redirect`", and are text files with the following format and properties:

- Lines that start with '`#`' or contain only white space are ignored.
- Each line otherwise contains one or more directory paths, separated by white space. If a directory path contains white space or other special characters, it should be double-quoted (i.e., as `"..."`).
- Multiple directories can be provided on a single line, or in different lines.
- Paths that are not rooted are taken as relative to the directory containing the redirect file.
- Paths that do not point to an existing directory are silently ignored.

When searching a directory, the directories found in a redirect file are also searched, in order, after the current directory. The search is recursive, so that arbitrarily deep hierarchies can be searched via the redirect file mechanism.

With redirect files, only the top directory of a hierarchy needs to be included in the search path (or given explicitly). This can be very convenient for organizing collections of native cell files, for example.

The **Path Files Listing** panel from the **File Menu** will list files found through the redirect files on separate pages for each redirected directory, just as for the directories contained in the search path.

2.8 Initialization Files

When *Xic* is started, a number of files are read. This section describes these files, and the order of access. None or these files is required to exist.

Prior releases of *Xic* could be configured to check for the availability of program updates on startup. There was also provision for display of a message if one was "broadcast" from the Whiteley Research web site. This latter feature was never used, and neither feature is currently supported in *Xic*. Thus, there is no longer a network access attempt on program startup, which may save time.

Program updates are handled in the help system (see 6.1.1), for all of the *XicTools* packages. Either the help system built into *Xic* and *WRspice*, or the stand-alone *mozy* program can be used to check for, download, and install updates. Giving the keyword "`:xt_pkgs`" will display a page that provides update information and download/install buttons.

If a new *Xic* release is run for the first time, the release notes will appear in a pop-up window, as if the **Notes** button in the **Help** menu was pressed. There is a file in the user's `.wr_cache` directory named `xic_current_release` that contains a release number. If, when *Xic* starts, this file is missing or the release number is not current, *Xic* will show the release notes and update the file. If the release numbers match, there is no action.

On installation, a default configuration is provided for *Xic*. The user will need to reconfigure *Xic* for their requirements. This reconfiguration is accomplished primarily by editing a custom technology file, which *Xic* reads on startup, and also by possibly setting some of the environment variables before starting *Xic*. These variables can be set in the user's shell startup file, as appropriate for the user's operating system.

The default technology file, plus several other files needed, are placed in a system-wide location on installation, usually `/usr/local/xictools/xic/startup`, which is included in library path. This directory is typically set to be read-only, thus the user must establish an alternative location in their own directory tree for customized startup files, and add this to the library path to the left of or instead of the default location. The default technology file provided with *Xic* is for generic MOSIS scalable CMOS.

X resource file

As the program starts and the graphics is initialized, the X window system may access various files for resource resetting. See the X documentation for details. The attribute (non-layer) colors used in *Xic* can be set through the resource mechanism (see A.10), but one must take care that these are not reset in the technology file.

`.xicinit` file

Next, an "`.xicinit`" initialization script, if present, will be read and executed. The user may create this file, it is not present by default. The initialization script uses exactly the same format as other script files, as are normally found along the script search path. The script can set user preferences or otherwise modify *Xic*. Since this file is read before other files, it can be used to set the search paths used to find other startup files, in particular the technology file. The base name for the script is "`.xicinit`", and the same extension as the technology files can be present.

If, for example, *Xic* is started with an extension "`.ext`" (-Text given on the command line), *Xic* will look for files `./.xicinit.ext` and `$HOME/.xicinit.ext`, then `./.xicinit` and `$HOME/.xicinit`, in that order, where "`$HOME`" indicates the user's home directory. The first file found will be executed. If *Xic* is started without a technology file extension, only the script files without an extension will be executed.

Technology file

If a technology file is being used, *Xic* will read the file at this point, before reading the user's script and macro files (below).

The technology file contains all of the information *Xic* needs for physical and electrical layout, extraction, and design rule checking, plus information on hard copy support, printer commands, and the like. It also provides values for a number of presentation attributes including the colors used on-screen.

The **Save Tech** button in the **Attributes Menu** creates an updated copy of the technology file in the current directory. Most of the changes to an existing technology file can be performed from within *Xic*, though some text editing may be required on occasion.

`.xicstart` file

Next, an initialization script, if present, will be read and executed. This file can be created by the user, is is not present by default. The initialization script uses exactly the same format as other script files, as are normally found along the script search path. The script can set user preferences or otherwise modify *Xic*, and, unlike the similar "`.xicinit`" file, performs these commands after the technology file has been read. The base name for the script is "`.xicstart`", and the same extension as the technology files can be present.

If, for example, *Xic* is started with an extension "`.ext`" (`-Text` given on the command line), *Xic* will look for the files `./.xicstart.ext` and `$HOME/.xicstart.ext`, and then `./.xicstart` and

`$HOME/.xicstart`, in that order, where "`$HOME`" indicates the user's home directory. The first file found will be executed. If *Xic* is started without a technology file extension, only the script files without an extension will be executed.

`xic_stipples` file
The `xic_stipples` file is read, which initializes the default fill pattern registers in the fill pattern editor in the **Attributes Menu**. Like the device and model libraries, the technology file, font files, etc., the library search path is used to locate this file. A default stipple file is provided, and new files can be obtained from the **Dump Defs** button in the **Fill Pattern Editor**.

`.xicmacros` file
Next, *Xic* will attempt to read a file with the base name "`.xicmacros`", and the same extension as the technology files can be present. This file does not exist by default, but is created if the user defines macro definitions which are mapped to key presses, as generated by the **Key Map** command in the **Attributes Menu**. The `.xicmacros` file is rarely if ever directly edited by the user.

If, for example, *Xic* is started with an extension "`.ext`" (-Text given on the command line), *Xic* will look for files `./.xicmacros.ext` and `$HOME/.xicmacros.ext`, then `./.xicmacros` and `$HOME/.xicmacros`, in that order, where "`$HOME`" indicates the user's home directory. The first file found will be read. If *Xic* is started without a technology file extension, only the script files without an extension will be read.

`.xic_font` file
If a file named "xic_font" is found in the library search path, the file is read to obtain the text font used for on-screen label text. This file is created by the user from the **Dump Vector Font** button in the **Font Selection** panel, and is subsequently editing to the user's requirements. The default font is hard-coded internally.

`.xic_logofont` file
If a file named "xic_logofont" is found in the library search path, the file is read to obtain the text font used for the **logo** (physical text) command. This file is created by the user from the **Dump Vector Font** button in the **Logo Font Setup** panel, and is subsequently editing to the user's requirements. The default font is hard-coded internally.

`xic_mesg` file
This is a text file providing the legal disclaimer. It once supplied text for the **About** window, but is no longer used for that purpose.

Device Libraries
As needed, *Xic* will also read the device library (`device.lib`) file, search and map the device models and help files, and open the first command line file for editing. The device library file supplies the device templates used in electrical mode. The model files provide SPICE models used for generating SPICE output. These files are read the first time access is required. Defaults are provided for these files, but the user will very likely need custom device and model library files.

2.9 Log Files and Error Reporting

There are several methods by which error and warning messages are presented to the user. In many commands, particularly those that use input from the prompt line, the prompt line is used to print messages informing the user of incorrect input, and general command status. These messages are intended to direct the user toward correct usage of the command.

More serious errors are reported in a pop-up window. There are two types of messages: those that are logged, and those that aren't. If a message is logged, it is assigned a unique sequence number, and is saved in the `xic_error.log` file discussed below.

The same pop-up window is generally used for both types of message. Most error and warning messages are logged. A few messages are unlogged, these generally report an immediate command failure due to some condition such as lack of a current cell, or something such as a help keyword not found message which is probably not worth logging.

The text window presenting an unlogged message will contain only that message. One of the disadvantages of unlogged messages is if several are emitted, only the most recent is shown in the window, the others are lost. This is unlikely to happen in current *Xic* releases.

The text window will display the sequence number and text of an emitted logged message, and some number (currently hard-coded as 20) of the previously emitted messages. One can scroll through the list to find previously emitted messages, which unlike in the unlogged case still exist.

The error message window contains two buttons in addition to the **Dismiss** button.

Save Text
This allows the user to save the text shown in the pop-up to a file. This may be useful for documenting errors seen for bug reporting, and for other purposes.

Show Error Log
This button will bring up a file browser window loaded with the `xic_error.log` file. This allows the user to browse all errors, in sequence. This can be used to revisit old errors that have scrolled off the end of the list in the pop-up error window.

2.9.1 Log Files

While *Xic* is running, various log files are produced. These files contain a record of operations and errors, which may be useful for debugging purposes. Ordinarily, though, many of the log files are rarely used, and these files are stored in a temporary directory which is removed when *Xic* exits normally. Other log files, such as DRC error reports, are saved in the current directory and are not removed on exit.

Below is a listing of the log files that are saved in a temporary directory. The files in this directory can be browsed from within *Xic* with the **Log Files** button in the **Help Menu**. In addition, a button in the error pop-up allows the `xic_error.log` to be viewed.

The **Logging Options** panel from the **Logging** button in the **Help Menu** selects whether or not certain operations are logged, such as those done during extraction. This will optionally produce additional log files not listed below.

`xic_run.log`
This file contains a listing of key press/release and mouse button press/release events, in a format which can be understood as script instructions. Although presently this feature in incomplete, the instructions can be used to "play back" the current session by executing the log file as a script. The file is limited in size to about 100Kb, at which point the file is given a "`.0`" extension and a new file is started. If *Xic* should ever crash or otherwise misbehave, the current `xic_run.log` should be included with the bug report sent to Whiteley Research. This will greatly help in tracking down the problem.

`xic_error.log`
This file contains a list of error messages generated during the session. The previous 20 errors are

displayed in the error pop-up window in *Xic*, but the `xic_error.log` file retains a complete record. This file may also be of use in diagnosing problems within *Xic*, and should be included with the bug report if it contains an entry relevant to the problem.

`xic_mem_errors.log`
This file, used under Unix/Linux only, is generated or appended to if memory corruption is detected. If this file exists when *Xic* exits, it will be emailed to Whiteley Research (by default). However, if either XICNOMAIL or XTNOMAIL is set in the environment, the file will instead be moved to the current directory, and a message will be printed requesting that the user mail it to Whiteley Research. Memory corruption should never occur, and this file contains stack trace information that will help identify the problem.

`read_cgx.log`
`read_cif.log`
`read_gds.log`
`read_oas.log`
`read_native.log`
These files contain messages emitted when a file is read into *Xic* for editing. The file name generated depends on the type of file read.

`write_cgx.log`
`write_cif.log`
`write_gds.log`
`write_oas.log`
`write_native.log`
These files contain messages emitted when a file is written to disk. The file name generated depends on the type of file written.

`convert_cgx.log`
`convert_cif.log`
`convert_gds.log`
`convert_oas.log`
`convert_native.log`
These files contain messages emitted when a file is converted directly to another format through the commands in the **Convert Menu**.

The size of the log files that grow progressively as *Xic* is running are size-limited to about 100Kb. If the file exceeds this size, the file is moved to the same name with a "`.0`" extension, and the original log file is reopened. Thus, a maximum of 200Kb per log of information is retained.

The environment variable XIC_LOGDIR can be set to an existing directory that will be used to store the log files. The log files will be placed in a directory

logdir/`xic`.*pid*

where *logdir* is the first defined of the environment variables XIC_LOGDIR, XIC_TMP_DIR, TMPDIR, or defaults to "/tmp". The *pid* is the process id of the *Xic* process. This directory is created when *Xic* starts, and is deleted when *Xic* terminates normally. If *Xic* terminates abnormally, the log files will still be around for inspection. If a user needs to look at a log file after running *Xic*, the file must be copied to another location before exiting *Xic*. The **!logfiles** command can be used to read logfiles from within *Xic*.

This mechanism lets multiple copies of *Xic* run on the same machine from any directory, and minimizes the pollution of the file system and in particular the current directory with a lot of generally unused log files.

2.9.2 Abnormal Termination Logging

If *Xic* experiences an internal memory referencing error, *Xic* will terminate. Such occurrences should be rare to nonexistent, however this is the ideal and generally not the reality. During a "panic", the following will happen:

- A subdirectory will be created in the current directory, with the name "`panic.`*pid*", where *pid* is the process id number of the running program.
- All cells in memory that have the modified flag set will be written into this directory. The files will be in the original file format. Cells created in *Xic* and never saved will be saved in native format. Although it can not be guaranteed that these files are not corrupted by whatever error occurred, generally they are clean and accurately reflect unsaved work. After a thorough check, they can be copied back to the original file name.
- A file named "`xic_panic.log`" is created in the current directory. This contains the log messages emitted while the modified cells are being dumped, and other information.
- The log files that are normally removed after normal exit are retained. The location of the log files is given in the `xic_panic.log` file.
- Unless either of the environment variables XICNOMAIL or XTNOMAIL is set, a stack trace is emailed to Whiteley Research, which will be analyzed to resolve the cause of the fault, and if possible the problem will be fixed in the next *Xic* release. The file that is emailed is named "`gdbout`". The file will be created in the current directory.

2.10 Plug-Ins

A "plug-in" is a software library that is read into a running program, that provides additional features or capability. Within *Xic*, plug-ins provide optional support for OpenAccess, and the Python and Tck/Tk languages. The plug-in provides an interface to external libraries that may or not be present on the user's computer. If the needed libraries are present, the plug-in will be loaded into *Xic* on program startup, and a message, such as

"`Using Tcl/Tk (tcltk.so)`"

will appear in the console among the text generated on program startup. If the needed libraries are not found, the plug-in is not loaded, but *Xic* will run normally except that the plug-in's features will be absent.

At present, plug-ins are supported on all platforms except for Microsoft Windows. Windows does not provide the type of shared library technology needed for plug-ins. Although a similar capability could be instituted, there are many substantial issues and it is not clear if it is worth the development effort.

Plug-ins are distributed as shared library code, and are found in the `plugins` sub-directory in the distribution area, i.e.,

prefix`/xictools/xic/plugins`

The plug-in files are version-specific, and will work **only** with the program from the same distribution file. Of course, *Xic* needs to be able to find its startup files for the plug-ins to be available. If *Xic* is not installed in the standard location, the XT_PREFIX environment variable should be set to enable *Xic* to find its startup files.

Normally, if a plug-in is not loaded, there is no message. If, however, the XIC_PLUGIN_DBG environment variable is set, diagnostic messages will be printed. These can help identify why the plug-in is not being loaded, and are instrumental in tracking down problems when the user expects success.

Lack of success loading a plug-in and generally due to the inability of the plug-in code to find the shared libraries needed on the host computer. Unless the library is "standard" on the system, which may be true of Python, then it will be necessary to use the LD_LIBRARY_PATH environment variable to specify where to look for the libraries. The libraries much match the address size (32 or 64 bit) of the running *Xic* program.

2.11 OpenAccess Support

This interface is presently not available under Microsoft Windows.

The OpenAccess plug-in is not provided with *Xic* packages, the user must build this from source, which requires OpenAccess source code.

OpenAccess is a semi-open-source database for CAD/EDA data. It is used by Cadence Virtuoso, Synopsys Custom Compiler, and by many other tools. It provides a commonality among tools from different vendors, and is intended to facilitate seamless integration of tools from different vendors into a process flow. OpenAccess is distributed by Si2 (`www.si2.org`). Source code and binary distributions are available for a number of operating systems, to registered users and coalition members.

Xic can connect to an OpenAccess (OA) database through a plug-in. Since there is no default location for OA, the user must set the `XIC_LIBRARY_PATH` or the `LD_LIBRARY_PATH` variable to include the library location in the search path during program loading. This is most conveniently done in the user's shell startup script.

Probably, the main interest in using OA is for limited compatibility with Cadence Virtuoso. There are two levels here. The first level is compatibility with the OA system. This is basically complete, as any *Xic* design can be saved to and read from OA without data loss or change. The second level is compatibility with the conventions and methods used in the Virtuoso product, much of which is proprietary or undocumented. This is a much tougher nut to crack. Presently, there is fairly reasonable capability of taking Virtuoso designs into *Xic*, but the reverse is not true. Presently, physical (layout view) data from *Xic* can be read by Virtuoso and should appear correct, however there is no netlist information or connection to a schematic. It is as if the layout view was read from a GDSII file. Schematic and schematic symbol views from *Xic* can not be read as anything but garbage by Virtuoso. There are plans for a data translation stage in the future to possibly adapt *Xic* schematics to Virtuoso format.

Likewise, The plug-in allows a direct interface to Synopsys Custom Compiler, and supports Python-based PCells including stretch handle and abutment protocols.

When the OpenAccess plug-in is loaded, there are several changes to *Xic*.

1. There is an **OpenAccess Libs** entry added to the **File Menu**. Pressing this will bring up the **OpenAccess Libraries** panel, which provides access to the existing OpenAccess design data.
2. A number of "bang" commands (text-mode commands that start with '!') are made available. These commands are typed into the prompt line to start. Much of the functionality of these commands is also available graphically in the panel.

!oaversion
!oanewlib
!oabrand
!oatech
!oasave
!oaload
!oadelete

In addition, the standard commands for reading and writing design data become operable with OpenAccess data. When specifying a cell, one provides two words: the OpenAccess library name and the cell name.

It is not possible to write to an OA library unless the library has been "branded" by *Xic*. By default, libraries created in *Xic* are writable from *Xic*, libraries created by other tools are not. The read-only status from *Xic* of any library can be set from the **OpenAccess Libraries** panel, or with the **!oabrand** command.

2.11.1 Representing *Xic* Cells in OpenAccess

When an *Xic* cell is saved in OpenAccess, up to three views may be created. The user has specified a library name where the views will be saved, and of course the cell name. Some write commands allow the user to save a cell under a different name.

If the cell contains physical data, this will be saved in a view named "`layout`" of OpenAccess view type "`maskLayout`". If the cell contains electrical data, the schematic will be saved in a view named "`schematic`" of view type "`schematic`". If a symbolic representation has been defined, this will be saved in a view named "`symbol`" of OpenAccess view type "`schematicSymbol`". This latter view can only exist, as part of an *Xic* cell representation, if a schematic view also exists. Reading or writing an *Xic* cell will involving translating each of these views that exist.

This group of properties applies to the OpenAccess interface.

stdvia property, number 7160
This property is applied to standard via sub-masters and instances, and is used by the translator to convert OpenAccess standard vias to *Xic* standard vias, and the reverse. The property is used in *Xic* to identify and specify standard via instances and sub-masters. The format of the property string is described in 5.8.1.

oa_cstmvia property, number 7161
This property is applied by the OpenAccess reader to master cells that represent a custom via. In *Xic*, vias are cells, they have no unique type as in OpenAccess. The string format consists of the cell identifier followed by parameter specifications. The cell identifier has the form

<libname><cellname><viewname>

This is followed by a space-separated parameter specification string in the same format as the pc_params property. A custom via master is basically a pcell sub-master.

When written back to OpenAccess, cells with this property will be ignored. A sub-master for the custom via will be created within OpenAccess when needed.

oa_orig property, number 7183
This property is used transiently when loading OpenAccess cell data into *Xic*. If is applied to cells,

and removed when reading completes. If an instance is read before the corresponding cell definition, a dummy *Xic* cell descriptor is created and given this property. The property string contains the library and cell names, separated by a forward slash ('`/`') character. Using this information, the cell is read later.

2.12 Python Support

This interface is presently not available under Microsoft Windows.

The Python (`www.python.org`) scripting language is a powerful, versatile language enjoying much popularity. In particular, it has become the language of choice for writing portable parameterized cells, as used in the PyCell Studio project from Ciranova, Inc. (now Synopsys). This download provides the essentials for creating portable pcells, using the Python language, and OpenAccess. Whiteley Research strongly favors this approach, and will integrate Ciranova standards as tightly as possible.

Python is made available, when Python-2.6 or newer is found on the user's computer, via the Python plug-in. Red Hat Enterprise Linux 6 and 7 provide a compatible native Python. Presently, only Python release 2.6 is supported on Red Hat Enterprise Linux 5, so installation of an updated package is required on that operating system.

The Ciranova PCell Studio provides Python 2.6, as well as OpenAccess. If using Ciranova, the Ciranova-supplied Python should be used.

Red Hat Enterprise Linux 6,7
: This supplies a native Python-2.6/2.7, which will work with the plug-in without any configuration. Unfortunately, this is not compatible with the Python-2.6 provided by Ciranova. *Xic* can use either one. The Python-2.6 provided by Ciranova was built with different setup flags for handling UTF8 text than the stock Python-2.6.

Red Hat Enterprise Linux 5
: The operating system provides Python-2.4, which is not supported. The Ciranova PyCell Studio provides Python-2.6, which is one source for a compatible Python. Another is to install the `python26` extension package. Using the **Package Manager** or `yum`, install

    `epel-release-5-4.noarch`.

  This will add additional repositories. Then, in the `epel` repository, find and install a release like "`python26-2.6.8-2.el5.x86_64`".

To use a non-default Python such as that supplied by Ciranova in the PyCell Studio, one will need to set the LD_LIBRARY_PATH variable to include the alternative shared library location. This will happen automatically if Ciranova's setup procedure is followed before starting *Xic* (see 5.6).

Failure to load the Python plug-in is by default silent. If the environment variable `XIC_PLUGIN_DBG` is set, diagnostics and error messages will be printed in the console when attempting to load plug-ins at program startup.

When the Python plug-in is loaded, *Xic* is able to execute Python scripts. This includes stand-alone scripts, and scripts that are used in parameterized cells. Note that Ciranova PyCells, which are also Python-based, are supported via OpenAccess, and are independent of Python support in *Xic*. However, future plans are to support PyCells natively in *Xic*. *Xic* is presently able to support the Ciranova protocols for stretch handles and abutment natively.

This information is preliminary, and may change.

The entire library of native script functions are callable from Python. However, at this point many of the more complicated data types found in the native function library are unsupported. There are two ways to call a native function from Python:

```
xic.native_func(args, ...)
xic.eval("native_func", args, ...)
```

The choice of style is up to the user, the first is probably slightly more efficient and is recommended.

The Python script must include some initialization lines in order to use the *Xic* interface. As a simple example, the script below will draw two boxes in the current cell, using the current layer.

```
import xic
import xicerr
import sys
sys.stderr = xicerr
xic.Box(2.0, 2.0, 6.0, 7.0)
xic.eval("Box", 1.0, 1.0, 5.0, 6.0)
xic.Commit()
```

The first line is mandatory for using any native script functions. It loads the *Xic* interface module.

The next three lines redirect Python error messages to the *Xic* error reporting system. These are optional, if not included Python messages will be printed on the console window.

The final three lines call functions from the native script library. The first two of these lines illustrate calling the `Box` function using the two syntax styles. The final line calls the `Commit` function, which registers the change with the undo system, among other things.

The first four lines are implicitly added during pcell evaluation, thus no not have to be included in a Python pcell script (see 5.1).

Presently, datatypes translate in the following manner. If an un-handled data type is encountered, the script will terminate with a fatal error.

Xic **type**	Python type
string	String.
scalar	Float.
array	List of float.
zlist	List of "`zlist`" followed by lists of six integers (LL, LR, YL, UL, UR, YU in internal units).
handle	A list containing "`xic_handle`", followed by the handle integer value. For stringlist handles only, the strings follow.

When these forms are passed back to *Xic* functions, they are reverted to the *Xic* data type. Note that handles can be passed through Python, but except for stringlist handles they are useless in Python at present.

When the Python plug-in is loaded, the **!py** command is available. This command will execute a script file containing Python commands, the path to which is given as the argument. Also, the following script functions are available:

`RunPython`	Run a Python script.
`RunPythonModFunc`	Execute a Python module function.
`ResetPython`	Reset the Python interpreter.

2.13 Tcl/Tk Support

This interface is presently not available under Microsoft Windows.

Xic provides a plug-in interface to Tcl/Tk. Tcl (Tool control language) is a popular open source scripting language, and Tk is a graphical package addition. The language syntax is provided in documentation supplied with Tcl/Tk, and is described in several books.

Since this capability is dynamically loaded, *Xic* can use this capability if it has been installed, but does not require the installation. Support is provided for Tcl, with and without Tk.

If Tcl/Tk have been installed via a standard distribution file on the system, which is common for Linux, the plug-in should be able to locate the shared libraries automatically. If the installation is non-standard, the user may need to inform the system dynamic linker of the shared library location. This is generally accomplished by setting the LD_LIBRARY_PATH variable in the environment, before running *Xic*. This would normally be done in the user's shell startup file.

There are two text-mode commands that can be used to run a Tcl/Tk script.

!tcl
This command will exist only if the Tcl language support plug-in is loaded, which will occur on program startup if the Tcl shared libraries are found. The script should contain only Tcl commands, not Tk.

!tk
This command will exist only if the Tcl and Tk language support plug-in is loaded, which will occur on program startup if both Tcl and Tk shared libraries are found. The script may contain any combination of Tcl and Tk commands.

In either case, the first argument is a path to a file containing the script body. Additional arguments are taken as arguments to the script. The script will be executed as if by the wish shell supplied with Tcl/Tk.

The startup file, which can be used to set defaults, is named "`.xic-wishrc`" in the user's home directory. The contents is analogous to the `.wishrc` file normally used with Tcl/Tk. The user must create this file if needed.

All of the *Xic* script functions are exported to Tcl/Tk and can be called by name from a Tcl/Tl script. However, only the basic data types are supported. There is also a function named "`xic`" which can be used in the following manner:

`xic` *function arguments...*

The function `xic` is a Tcl function which loads the interface function or user-defined function given in the first argument (a string). User defined functions can be accessed if they are already known to *Xic*, i.e., they were defined in a library file or were defined in a previously-run *Xic* script. The arguments to the function follow, and should match the arguments expected by the function. This form must be used when executing a user-defined function.

The variable type of an argument is inferred as follows:

- A single-token numeric value without leading or trailing characters not part of the number is taken as a scalar.
- A token of the form &*arrayname*() is taken as an array.
- Anything else is taken as a string.

To explicitly coerce a numeric token into a string, backslash escaped double quotes should be used to delimit the token. For example, `\"1.234\"` is taken as a string. The backslash prevents tcl from removing the double quotes before passing the token.

Arrays passed to interface functions must use "0", "1", etc. as indices, and are ordered accordingly (in tcl, array indices can be any text token and have no natural order). The "0" element (at least) must be set before the array can be passed to a function. If the array is dynamically expanded, new tcl elements will be created. The initial size of the array is implied by the largest contiguous index assigned. Thus, for example, if the interface function requires an array of size 4, the following tcl code could be used

```
set array(0) 0
set array(1) 0
set array(2) 0
set array(3) 0
xic Function &array()
```

When the function returns, the array values will be updated. Only one-dimensional arrays are available.

There is an additional special tcl function which has been added.

```
xwin win_name
```

This function returns the X window id of the tk window given as a widget path in *win_name*. This is used to obtain the window id of a tk window to be used for *Xic* graphics through the `GRopen` interface. A suggested way to use a tk window for exported drawing from *Xic* is given in the example below. The `xwin` procedure is used to obtain the window id. This window should be configured with '`-background ""`' which allows redraws to be handled through a procedure bound to the window with the `bind` command which responds to expose events. Otherwise, expose events will cause the window to be redrawn in gray *after* the event handler is called. A pixmap is used to store the image for redraws.

Example

```
# This is the window used for drawing by Xic.
# Note the '-background ""' directive.  This
# is necessary for proper redrawing after expose
# events.
frame .f -width 8c -height 8c -background ""
pack .f

set win_id [xwin .f]
```

```
set ghandle [xic GRopen ":0" $win_id]
# The win_id is the X id of the drawing window,
# the ghandle is the handle value returned from
# Xic upon opening graphics on this window.

set size(0) 0
set size(1) 0
set size(2) 0
set size(3) 0
xic GetWindowView 0 &size()
# The size array contains the displayed area of the
# cell in the main Xic window, in order L, B, R, T

xic GRdraw $ghandle $size(0) $size(1) $size(2) $size(3)
# This draws the Xic view into the Tk window

xic GRupdate $ghandle
# Due to the way Tk (and X) works, unless GRupdate is
# called after drawing, the drawing won't be visible.
# The operations are stuck in a cache somewhere waiting.
# GRupdate flushes the operations.

set dsize(0) 0
set dsize(1) 0
xic GRgetDrawableSize $ghandle $win_id &dsize()
# The dsize array contains the size in pixels of the
# Tk drawing area.

set pixm [xic GRcreatePixmap $ghandle $dsize(0) $dsize(1)]
xic GRcopyDrawable $ghandle $pixm $win_id 0 0 $dsize(0) $dsize(1) 0 0
xic GRupdate $ghandle
# We have created a pixmap of the same size and depth as
# the drawing area, and copied the drawing area into it.
# This will be used to redraw the drawing area after an
# expose event.

bind .f <Expose> {
    # This sets up a handler for expose events.  Expose
    # events are received when a previously obscured part
    # of the window is uncovered.  The pixmap is copied
    # into the Tk window.
    xic GRcopyDrawable $ghandle $win_id $pixm 0 0 $dsize(0) $dsize(1) 0 0
    xic GRupdate $ghandle
}
```

The TextCmd script function can be used to launch a tcl/tk script. At present, tcl/tk scripts are not recognized in the script path, but one can use a native language wrapper to include tck/tk scripts in the **User Menu**.

The following native script functions can also be used to run Tcl/Tk scripts, or perform other related manipulations related to the Tcl/Tk interpreter.

`RunTcl`	Run a Tcl or Tk script.
`ResetTcl`	Reset the Tcl/Tl interpreter.

This page intentionally left blank.

Chapter 3

Graphical Interface, Commands and Operations

Figure 3.1 shows a view of the *Xic* graphical user interface. There is generally a single large window present when *Xic* first starts. The window can be repositioned, and the size of the window can be adjusted through the window manager methods.

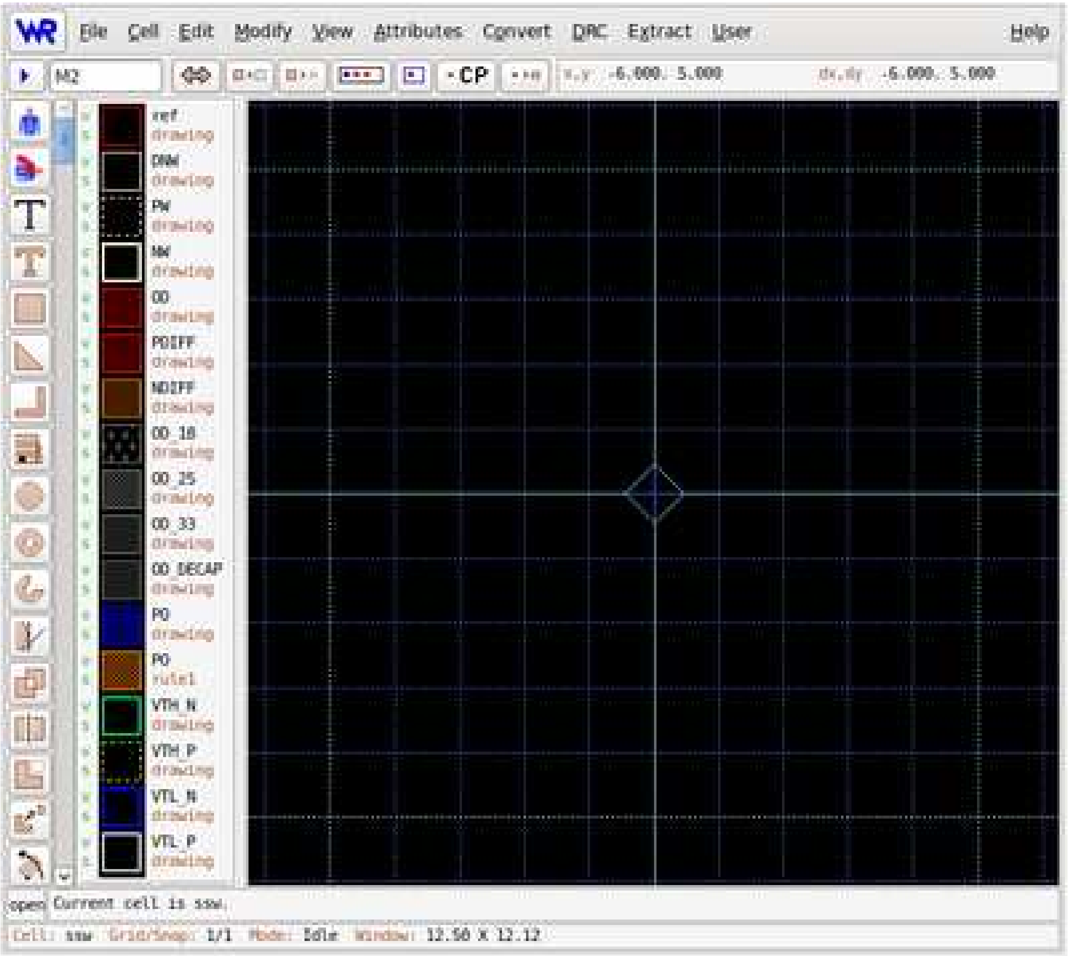
The column of buttons along the left is the "side menu" and is visible when the current cell is being edited. To the right is the scrollable layer table, which displays the layers supported by the process. If the XIC_MENU_RIGHT variable is set in the environment when *Xic* starts, the layer table and side menu will be located along the right of the window. If the XIC_HORIZ_BUTTONS environment variable is set, the "side menu" buttons will actually be arrayed across the top of the window. The side menu is only displayed when editing. The layer table may also be invisible, as the user has this option.

The "top menu" contains buttons and other controls and displays, located near the top of the window, below the main menu bar. The prompt line, where the user interacts textually, is just below the main drawing window. To the left of this is the "keys pressed" area. Below this is the status line, which displays information about the program state.

These features will be fully described in the sections that follow.

Xic has eleven drop-down menus, arrayed in a menu bar which extends across the top of the main application window.

Figure 3.1: Default *Xic* screen layout.



File Menu	Commands to open, save, and list files and cells. This menu also contains the printer interface.
Cell Menu	Commands to access and manipulate the database of cells in memory.
Edit Menu	Commands which are used to modify the current design.
Modify Menu	Supplemental commands for layout modification.
View Menu	Commands which affect the presentation of the current design, including the selection of physical and electrical (schematic) modes.
Attributes Menu	Commands which affect the presentation of the design, such as the colors used.
Convert Menu	Commands for importing and exporting designs to various non-native file formats.
DRC Menu	Commands associated with design rule checking.
Extract Menu	Commands associated with the extraction of electrical information and netlists from the physical layout, and layout versus schematic checking.
User Menu	The script debugger, and the buttons that correspond to user-generated scripts.
Help Menu	Documentation and the entry into the help system.

If the mouse button is stationary over a menu button for a second or two, a "tooltip" will appear. This is a transient window that contains a sentence describing the function of the command. This also provides the internal name for the command. Every command has an internal name of five characters or fewer. This name can be used as a keyboard accelerator, and as back-door input to the help system. The help keyword for the command is "`xic:`" followed by the command name, for example "`xic:prpty`". Typing a question mark ('`?`') into *Xic* followed by the keyword will display the help text for the command.

3.1 Prompt Line

The prompt line is a single-line dialog box just below the main drawing window. Messages and prompts are displayed in this area, as well as textual input to *Xic*.

The prompt line has two operating modes. In the normal mode, text is read-only. Messages appear on the prompt line to provide information and feedback in many commands. This is "non-editing" mode.

In non-editing mode, text can be selected by dragging with button 1 held down. Selected text is available for export to other windows, as the primary selection (see 3.13.3).

The prompt line can handle more text than is visible in the display area. If a string is longer than the display area, initially the rightmost part of the message string will be shown. Clicking in the prompt area with button 1 near the left border will show the start of the string. Clicking in the prompt area near the right border will show the end of the string. Clicking in the interior of the prompt area will show the middle part of the string, proportionate to click location.

3.1.1 Prompt Line Editing

Some commands will convert the prompt line to editing mode. In this mode, the background color changes, and text typed by the user will appear in the prompt line window. Keys pressed when the main window has focus are directed to the prompt line.

When editing, the behavior is slightly different depending on whether the mouse pointer is over the prompt line area, or not. This is (or should be) true whether or not the window manager is click-to-focus or focus-follows-mouse. When the mouse pointer is over the prompt line, which gives the prompt line complete focus, the prompt line background color may be different from when the pointer is elsewhere. When the pointer is elsewhere, but the main window has focus, key presses are still sent to the prompt line, but there are a few keys, such as the arrow keys, which will operate on the drawing window rather than the prompt line.

When prompt line editing starts, the mouse pointer is "warped" to the left edge of the prompt line, providing full focus automatically. With the mouse pointer over the prompt line:

1. The **Numeric Keypad +** and **Numeric Keypad -** keys will send a normal + or - character and not zoom the drawing window display.

2. The arrow keys will move the prompt line text cursor, or perform some other operation specifically for text editing, depending on the command. These will not pan the display.

With the mouse pointer not over the prompt line, the keys mentioned will have their normal zoom and pan functionality. In text edit mode, key bindings from the table below are available, provisionally for the arrow keys as explained.

Prompt Line Editor Bindings

Ctrl-a	Move cursor to beginning of line
Ctrl-e	Move cursor to end of line
Ctrl-k	Delete to end of line
Ctrl-p	Paste primary selection at cursor
Ctrl-u	Delete current line
Ctrl-v	Paste clipboard at cursor
Left	Move cursor left one character
Right	Move cursor right one character
Page Down	Move cursor to right by half a line, scroll if necessary
Page Up	Move cursor to left by half a line, scroll if necessary
Backspace	Delete previous character
Delete	Delete next character
Esc	Exit editing, abort operation
Enter	Terminate editing

The **Backspace** key deletes the character or hypertext reference to the left of the cursor and moves the cursor to the left, and **Delete** deletes the object at the cursor. **Ctrl-u** deletes the entire line. **Ctrl-k** will delete the character at the cursor and all characters to the right. **Ctrl-a** and **Ctrl-e** move the cursor to the beginning or end of the line, respectively. The line will scroll to the left or right if longer that the available space, when the cursor hits the left and right boundaries. The **Esc** key exits edit mode, discarding the input. The **Enter** key exits edit mode, saving the input. The cursor can be at any position when **Enter** is pressed.

Double-clicking with button 1 in the prompt line area will effectively send an **Enter** character, terminating editing. Note that a double click requires two rapid clicks, if too slow two single-click events will occur.

Special characters can be entered using the Unicode escape **Ctrl-u**. The sequence starts by pressing **Ctrl-u**, then entering hex digits representing the character code, and is terminated with a space character or **Enter**. The Unicode coding can be obtained from tables provided on the internet, or from applications such as KCharSelect which is part of the KDE desktop. These are generally expressed as "`U + xxxx`" where the `xxxx` is a hex number. It is the hex number that should be entered following **Ctrl-u**. For example, the code for π (pi) is `03c0`. Note that special characters can also be selected and copied, or in some cases dragged and dropped, from another window.

There is no limit on the number of characters in the string, which can be much longer than the display space. The **Page Down** and **Page Up** keys move the cursor to the right or left (respectively) by half the number of characters displayable in the prompt area, and will scroll if necessary to keep the cursor visible.

The **Ctrl-p** and **Ctrl-v** keys paste text from the primary selection and clipboard, respectively, at the cursor. Under Windows, these actions are identical, text is obtained from the Windows clipboard. Under Unix/Linux, clicking with button 2 will also paste the primary selection, and button 3 will also paste the clipboard. The primary selection is generally the most recently selected text in any window, the clipboard contains text that was explicitly saved via an operation in a text entry window.

While in editing mode, the keypress display to the left of the prompt line is replaced with two or three buttons. The **R** and **S** buttons, which are always present when the prompt line is in editing mode, provide access to five general-purpose registers for text, plus a register for the "last" text. Both buttons produce a drop-down menu containing register numbers. If a selection in made in the **S** menu, the text currently in the prompt area is saved to the register whose number was selected. Any previous content is overwritten. If a selection is made in the **R** menu, text saved in the register whose number is selected will replace the text in the prompt area. The saved text can contain hypertext entries (see below).

In some contexts, a third ("**L**") button appears. This provides access to the "long text" capability, which allows multiple lines of text to be entered by providing access to a text editor window.

When editing mode is exited, the buttons disappear and are replaced with the keys pressed display. If **Enter** was pressed to terminate editing mode, the text is automatically saved in register 0, and will be available from the **R** menu the next time editing mode is entered.

For some property strings, if a line of text that is longer than 256 characters is opened for editing on the prompt line, the **Text Editor** will appear, loaded with the text. The text will be saved as a "long text" item.

These features are described in more detail in the description of the **label** command in 7.9.

Non-printing characters in the text will be displayed using special symbols, which can be edited (in edit mode) as normal characters. The non-printing character most likely to appear (and the only one that probably should appear) corresponds to the line termination character. These cause a line break when the text is displayed as a label on-screen, and can be entered while in editing mode with **Shift-Enter**. In Windows, these are shown as a paragraph symbol, while in Unix/Linux a "v/t" (vertical tab) glyph is used. Other characters will show as a black dot in Windows, or a "strange" character in Unix/Linux.

The prompt line participates in the drop protocol for files. Files dropped on the prompt line in normal mode have the same effect as files dropped in the main drawing window - the file will be taken as layout input and displayed in the drawing window.

When in text editing mode, files dropped in a drawing window or the prompt line will not be displayed, rather the full path to the file is inserted into the text line at the cursor. This means that when responding to a prompt to open a file, the **File Selection** pop-up from the **File Select** button in the **File Menu** can be used to find the file. The file can then be dragged into the main window or the prompt line window and dropped, and the name will appear on the prompt line. Also while the prompt line is in editing mode, pressing the **Open** (green octagon) button or the **Open** menu entry of the **File**

Selection pop-up will load the selected file path into the prompt line rather than opening the cell for editing. In most situations where *Xic* prompts for a file path via the prompt line, a simplified version of the **File Selection** pop-up will appear while editing is active.

3.1.2 Hypertext

Xic contains a "hypertext" capability, which is active in electrical mode. By default, the names of circuit nodes and devices are internally assigned, implying that the name of a particular device or node name of a particular wire net might not be well defined. This poses a problem when one wishes to identify a specific device or wire net by name. The hypertext feature addresses this issue, as do the node name mapping and `name` property assignment features.

This is necessary, for example, when setting device properties which reference other devices or nodes. The device names and node numbers might change, thus property text could become invalid if it were static. Instead, internally, strings are stored as data structures which reference pure text as well as devices and nodes by internal reference. Thus, these hypertext strings are always valid.

One creates a hypertext reference by clicking on the schematic while text input is being solicited in the prompt area. The returned data can be a node reference, a device branch reference, or a device name. The string, as currently defined, is inserted into the displayed text in the prompt area in color. Note that one can only delete the whole item with the **Delete** and **Backspace** keys, the hypertext references are treated as single items.

¡p¿ One will encounter hypertext when using the prompt line editor as itemized below. In these cases, one could type in the text, however if due to future modifications that text changes, the present text would be wrong. If is therefor advantageous to use hypertext, by, e.g., clicking on a device in a drawing window rather than typing its name.

- When creating text for properties applied to electrical devices and circuits, for referencing other devices and nets in a name-independent manner. This applies when adding or editing properties from the **Property Editor** provided by the **Properties** button in the **Edit Menu**, and when subsequently editing the label text (and underlying property) using the **label** button in the side menu.
- When creating labels that require reference to devices or nodes, such as using spicetext labels to add such things as `.measure` lines.

  When creating a label, clicking on a connection point in the drawing, for example, will enter a hypertext link to the node into the label. The hypertext is shown in a different color in the prompt line. The label will always display the correct name for the node, should the name subsequently change. This is the means by which node labels can be added to the drawing.
- When selecting nodes and branches to plot, after simulation. The reference points selected by clicking are all hypertext.

There are three types of reference that can be defined by clicking in a schematic.

Node Reference
: If the user clicks over a wire or on a contact point of a device or subcircuit, a node reference is established. The colored hypertext entered into the prompt line as a response is of the SPICE form "`V(`*name*`)`", where *name* is the node name, which is an integer by default. The string, when printed or shown as a label, will always show the correct name for the node selected.

"Hidden" target
: Some devices have a "hidden" target, which is usually shown as a '+' symbol as part of the device schematic representation. The hidden targets are defined in the device definition in the device library file, so that the meaning and location may differ. In the default device library, most two-terminal devices have such a point, which generally returns a branch node or function which specifies the current through the device. For Josephson junctions, the target represents the junction phase. Clicking on this point in a drawing window will insert the corresponding reference.

Name Reference
: Clicking within the bounding box of a device or subcircuit, but not over a node or hidden target, will insert a name reference. The returned text is the name of the instance, as derived from the `name` property attached to the device or subcircuit. This can be applied by the user, to give the device a fixed name. If no `name` property is applied by the user, *Xic* will generate one with an internally generated name.

The node references and hidden targets are also the sensitive points when using the **plot** and **iplot** commands.

Note that these targets are active at any level of the hierarchy. However, they are generally not selectable unless the containing subcell is shown expanded as a schematic. If a subcircuit is shown as a symbol, one can still select internal points for hypertext references by using a proxy window. This is described in the next section.

This feature can be used to set up specialized SPICE output. Suppose one wishes to use a `.save` line in *WRspice*. A `spicetext` label can be created, where the nodes to be included in the save are inserted in the label by clicking on the drawing. The resulting `.save` line will always save the clicked-on nodes, whether or not the actual node names change.

For another example, suppose one needs to apply a functional dependence to a voltage source in the circuit to the voltage of some node. One would accomplish this with the following procedure.

1. Open the **Property Editor** and use the **Add** menu to initiate addition of a `value` property.
2. In the prompt line, type the equation representing the desired functional dependence, and whenever the node voltage text is needed, click on that node in a drawing window.
3. Press **Enter** to complete the operation.

The equation should appear in the property label near the voltage source. This could be, for example, "`2*v(4) + v(5)`", if default node names are used. Later, after modifying the circuit, one might notice that the label now reads "`2*v(6) + v(8)`". The internal node numbering has changed due to the modification, but the source still references the correct circuit nodes. This would not be the case if ordinary text was used for the equation string.

3.1.3 Proxy Windows

If one presses the **Ctrl** and **Shift** keys while clicking with button 1 on a subcell, a sub-window will appear, containing the content of the subcell. This works in electrical and physical mode, while the prompt line editor is active and not.

In electrical mode, the sub-window will display the master as a schematic, whether or not it is set to display symbolically. The sub-window, in this case has the important feature that it is a proxy for the main window for hypertext. When using the prompt line editor, clicking in the sub-window can add hypertext references to the prompt line, just like clicking in the main window.

One can also hold **Ctrl** and **Shift** and click on an instance in a proxy window, which will produce a new proxy window showing the master of the clicked-on instance. One can repeat the procedure to any depth, however at present there are only four sub-windows available, and windows will be reused if the depth exceeds four.

When a sub-window is active as a proxy, a label is displayed in the window menu bar. This will give the “proxy path” which consists of one or more subcircuit names, separated by periods. These are the subcircuits clicked on, up to the top level. The sub-window otherwise behaves normally, and one can switch to view another cell, or go to physical mode. The proxy label will disappear, and the sub-window will no longer act as a proxy. If one returns to viewing the original cell, the label and capability will return.

3.2 Keypress Buffer

To the left of the prompt line is the key press buffer area. This area displays the last five keys typed into the main drawing window. The keypress buffer remembers up to 16 characters, though only the last five are shown. It is cleared when **Esc** or **Ctrl-u** is typed. If the key sequence in the buffer uniquely prefixes a menu command, the command name is displayed, and the command is executed. The command names are a short mnemonic, displayed in the “tooltip” that appears when the pointer rests over a command or menu button.

Most commands have at most five characters in their command name, the exceptions are the scripts in the **User Menu**. For these, the menu text is the same as the command name, and it may take more than five characters to uniquely define the command.

The keypress buffer can be forced to literally match menu items by typing **Enter**. Consider the two entries in the **User Menu**: **spiral** and **spiralform**. Typing “spiral” does nothing, as this is a prefix of both entries. In order to run spiral by typing the command prefix, type “spiral” then **Enter**. This works for any menu commands where one entire command is a prefix of another.

When the prompt line is in editing mode, i.e., a command is active that requires user text input, the keys display is replaced by buttons associated with the editing function. The key press display returns when editing mode is exited.

Each drawing window (main window and the sub-windows produced with the **Viewport** button in the **View Menu**) has its own keypress buffer, and matching commands will apply to the window into which the text was typed, if applicable. In sub-windows, the key press buffer displays in the menu bar area, to the right.

3.3 Quoting

When giving input to *Xic*, single and double quotes can be used to “hide” characters, such as space characters, that *Xic* would otherwise interpret incorrectly. *Xic* will generally strip the outermost quotes before processing, so inner-level quotes will be retained (quote marks of different types nest). A quote mark which is preceded by a backslash will be treated as an ordinary character.

As an example, consider the prompt of the **Open** command. The command prompt expects one or two tokens. The first token is the name of a file to open. The second token, if given, is the name of the cell to edit if the first token names a multi-cell file such as a GDSII file. Suppose that the file is in a directory named "`Xic Files`". Without the quoting mechanism, there is an obvious problem. To edit the file, one enters, for example (each of these would work),

```
"Xic Files"/my_design.gds
"Xic Files/my_design.gds"
Xic" "Files/my_design.gds
```

The double quotes make each of these strings appear to *Xic* as a single word.

3.4 Menu Selection and Accelerators

Menus from the main menu bar are displayed when the left mouse button (button 1) is pressed over a menu bar entry. The drop-down listing of entries will appear. A selection can be made by releasing the mouse button over the item to be selected. Alternatively, clicking the mouse button will also cause the menu to appear, and clicking over the menu will select the item under the pointer, and retire the menu. While the menu is visible, keypresses are "grabbed" by the menu, and so will not be sent to other windows or applications. While a menu is visible, the up and down arrow keys will cycle through the menu entries, highlighting each in sequence. Pressing **Enter** will "press" the highlighted entry. The entries in the side menu are mostly toggle buttons, which are activated by clicking with mouse button 1.

Commands can also be executed by typing an accelerator while the mouse pointer is in a drawing window. Commands can be exited by selecting another command in most cases, or by pressing the **Esc** key. Some commands are switches which remain in effect until selected again.

There are multiple accelerator functions available.

1. **Alt**-*char* brings up the menu keyed by *char* where *char* is the character that is underlined in the name in the menubar. If this is followed by a character underlined in one of the menu entries, that function is invoked. For example, typing **Alt-fp** (press and hold **Alt**, press **f**, release **Alt**, press **p**) engages the **Print** command in the **File Menu**.
2. If the menu entry has something in the second column, that is also an accelerator. For example, in the **File Menu**, the **Quit** entry has "Ctrl-Q" listed in the second column. This indicates that pressing **Ctrl-q** will invoke the **Quit** command. The menu doesn't have to be visible.

   Under Unix/Linux, the menu accelerators can be changed interactively. Click on a menu to open it, then move the pointer over one of the entries (it will be highlighted). Pressing **Shift**, **Ctrl** or **Alt** along with another key will add that accelerator to the menu entry, or change an existing accelerator. With the menu invisible, entering that key combination will "press" the assigned button, unless the combination happens to be used elsewhere for another purpose (it must be unique in the menus, at least). Under Windows, the menu accelerators can not be changed.
3. Every command has a name, shown in the tooltip bubble that appears after the pointer is stationary over the button for a second or two. Typing the first few characters of this name will trigger that command. Only the characters required to uniquely specify the command name among all commands currently is scope are required. When activated, the name of the command is printed in the key press buffer window. For example, "`pus`" triggers **Push**.
4. One can define macros for keypress combinations with the **Define Macro** command button in the **Attributes Menu**.

3.5 Keyboard Input

The main window must have the keyboard focus in order for *Xic* to receive keyboard input. Under some window managers, including under Windows, the frame of the main window can be clicked on to give that window the focus, and the focus will remain with that window regardless of the location of the pointer. In other cases, the pointer must be in the main window in order to give the main window the focus.

If a command is active that is prompting for input, the keystrokes will appear on the prompt line, the key press display will be replaced with buttons, and the prompt line background will appear in a lighter color. See 3.1.1 for a description of the key bindings that are in force while in editing mode.

If not in editing mode, the characters will be added to the buffer displayed in the keys area. After each character is added to the buffer, the buffer is compared with all menu command names, and if the buffer uniquely matches the first characters of a menu button name, that button will be activated. Only a few characters can be saved in the buffer, and after the buffer is full, keystrokes will be ignored. The buffer can be cleared with **Ctrl-u** (hold the **Ctrl** key and press **u**). The buffer is also cleared after each command match, although the display will show the full name of the command. The **Backspace** key will delete the last character entered. There are other accelerators for most menu commands.

The '!' character will switch the prompt line to editing mode to solicit one of the text-mode commands. The '?' character will switch the prompt line to editing mode to obtain a help keyword or directive. There are many other keys with special significance to *Xic*, summarized in the table below. These keys should be memorized by the user, as there is no alternative way to invoke their function.

Character	Result
!	Enter text-mode command
?	Enter help keyword, URL, or path to image or HTML file
Esc	Exit current command, or deselect selections
Tab	Undo operation
Shift-Tab	Redo last undone operation
Delete	Delete selected objects
Forward Slash	Swap current transform with saved (Register 0) transform value.
Arrow Keys	Pan
Shift-Arrow Keys	Fine pan
Ctrl-Arrow Keys	Cycle rotation and mirror transformations
Numeric +	Zoom in, expand by 2
Shift-Numeric +	Zoom in by 10 percent
Numeric −	Zoom out, shrink by 2
Shift-Numeric −	Zoom out by 10 percent
Home	Center full view cell
Page Down	Show next DRC error in **Show Errors** command
Page Up	Show previous DRC error in **Show Errors** command
Ctrl-a	Select associated labels
Ctrl-c	Interrupt
Ctrl-e	Enter coordinate
Ctrl-g	Change grid
Ctrl-k	Delete-to-end when editing
Ctrl-n	Save view
Ctrl-p	Deselect associated labels
Ctrl-r	Redraw window
Ctrl-u	Clear input buffer
Ctrl-v	Print program version
Ctrl-x	Expand cells
Ctrl-z	Iconfiy

Just as the '!' character switches the prompt line to editing mode to accept a command (see 19, the '?' character will switch to editing mode, to accept a "help directive".

A "help directive" can be one of the following:

- A help system keyword, so "? *keyword*" is the same as "`!help` *keyword*", i.e., a shortcut to the **!help** command. If no *keyword* is given, and the program is in a command mode, meaning that the **Mode** entry in the status line is something other than "`MAIN`", then the help shown will apply to the current mode. Otherwise, the default help topic is shown, as for "`!help`" without arguments.

- A general URL or path to a compatible local file. The help window will display the file or URL, if possible. In particular, image files can be displayed this way. A URL must be complete, including the "`http://`" prefix. Most web sites use style sheets and other constructs not handled by the simple rendering engine in the viewer window, so it is not great for general web-surfing, but it may be good enough for some purposes.

- One of the single character directives. These apply only after '?', and print information that is not from the help system, but derived from internal tables. These are given in the table below.

Character	Result
!, b, B	Giving exactly one of these characters will print a listing of the '!' commands that are available in the program.
v, V	Giving exactly one of these characters will print a listing of the variable names that have significance within the program. Variables are listed whether or not the variable is actually set.
s, S	Giving exactly one of these characters will print a list of variables that are currently set, the same as the **!set** command without arguments.
f, F	Giving exactly one of these characters will print a list of all of the internal script interface functions available within the program.

Each listing will provide the listed items as colored links. Clicking on the links will pop up help about the item.

The *Xic* program is modular, and the *XicII* and *Xiv* virtual programs are effectively *Xic* with only a subset of modules. The listings provide definitive summaries of the functions and variables actually available in the feature set, in case this is not clear from the documentation.

The **Esc** (Escape) key terminates any command and clears the key press buffer. Many commands can also be terminated by pressing the command button a second time, or by selecting a new command. After pressing **Esc**, the mode listed in the status area should be "`MAIN`".

If pressed in idle mode, all selected objects will be deselected.

The **Tab** key performs an **Undo** command, which will undo the last operation, and has the same effect as pressing the **Undo** button in the **Modify Menu**. Pressing the **Shift** key along with the **Tab** key will instead redo the last undone operation, which is the same as pressing the **Redo** button in the **Modify Menu**.

Pressing the **Delete** key will delete any objects currently selected. Objects in a drawing can be selected with button 1 operations (see 3.6.1). This has the same effect as the **Delete** button in the **Modify Menu**. If the **Rulers** button in the **View Menu** is active, the **Delete** key will delete rulers and not other objects.

Outside of any command, pressing the forward slash '`/`' key will swap the current transform with the saved transform in Register 0, if any. The saved transform is the one in force for the last move or copy operation. The current transform is also saved when terminating a **Move**, **Copy**, or **Place** command.

Without the **Ctrl** or **Shift** keys pressed, the arrow keys will pan the display in the drawing window which contains the pointer by one-half screen in the direction of the arrow. If **Shift** (but not **Ctrl**)is held while pressing the arrow keys, the display will instead pan by ten percent. Panning can also be performed with the middle mouse button (button 2), and with the mouse wheel.

Holding **Ctrl** (but not **Shift**) while pressing the left and right arrow keys will cycle the current rotation setting, otherwise set with the **xform** command in the side menu. This affects moved and copied objects and new instances.

Holding **Ctrl** (but not **Shift**) while pressing the up arrow key will toggle the current **Reflect Y** state of the **Current Transform**.

Holding **Ctrl** (but not **Shift**) while pressing the down arrow key will toggle the current **Reflect X** state of the **Current Transform**.

Holding both **Shift** and **Ctrl** while pressing the left or right arrow keys will cycle through the previous views in the window which has keyboard focus. This is similar to the **prev** and **next** menu commands in the **View** command of the **View Menu**. The last five views of a cell are saved.

Holding both **Shift** and **Ctrl** while pressing the up or down arrow keys will increment or decrement the subcell expansion depth, as if giving a '+' or '−' to the **Expand** pop-up, affecting the drawing window that has the keyboard focus.

The arrow keys may have special functions in individual commands, which override the behavior above. This is noted in the descriptions of the commands.

The + and − keys in the numeric keypad area will zoom the display in or out by a factor of two, respectively, in the drawing window where the pointer was located at the time of the key press. The action is similar to the **Zoom** command in the **View Menu**, and the button 3 operations. On some systems, these keys must be defined using the mapping facility provided by the **Key Map** button in the **Attributes Menu**.

If the **Shift** key is held while pressing the numeric keypad +/− keys, the zoomin/zoomout factor is reduced to 10%.

Pressing the **Home** key will center and fully display the current cell, in the window where the pointer was located at the time of the key press. This can also be done with the **View** command. On some systems, this key must be mapped with the **Key Map** command in the **Attributes Menu** in order for this functionality to be available.

The **Page Up** and **Page Down** keys are used with the **Show Errors** command in the **DRC Menu**. **Page Down** will show the first and subsequent errors. **Page Up** will show the previous error(s). Pressing **Ctrl-f** will have a similar effect to **Page Down**, and either **Ctrl-b** or **Ctrl-p** will simulate a **Page Up** press. On some systems, the **Page Up** and **Page Down** keys must be mapped using the **Key Map** command in the **Attributes Menu**.

The command line interface through the prompt area provides an interface to operating system commands, as well as to a number of internal commands which are often rather specialized and not associated with a menu button. Each of these commands starts with an exclamation point ('**!**'), and may be entered when no other command is active, or inside of many commands. These key presses are not recorded in the "keys" area below the side menu. If the command entered matches one of the internal commands, that command is executed. Otherwise, an operating system shell and associated window is produced to execute the command, with the exclamation mark stripped. If the '**!**' is followed immediately by **Enter**, an interactive subshell window is brought up. See Chapter 19 for a listing of the '**!**' commands.

The keyboard function keys, usually labeled F1 – F12, can be mapped by the user to provide an alternate means of pressing buttons in the menus. The mappings are added to the technology file with a text editor, following the syntax described in A. These mappings are completely up to the user to define, and no default mapping is installed (though the supplied technology file contains a mapping).

There are several control characters (characters entered while holding the **Ctrl** key) which perform operations in *Xic*. These are hard coded, and are in addition to any accelerators listed in the drop-down menus from the main toolbar. These are also in addition to accelerators from pop-up windows that have accelerators in their menus. These control keys supersede a menu accelerator using the same key.

Ctrl-a

In electrical mode, outside of any command, pressing **Ctrl-a** will cause the associated labels of any selected device or wire to also become selected. If labels are selected, then pressing **Ctrl-a** will cause their associated device or wire to also become selected. The associated labels can be deselected by pressing **Ctrl-p**. This is useful for determining which labels are associated with a given device or wire, and *vice-versa*.

When entering text to the prompt area, **Ctrl-a** will move the cursor to the beginning of the line.

Ctrl-c

This key sends an interrupt signal to *Xic*. When an interrupt is received, and *Xic* is performing a lengthly operation, the user is generally given the option of aborting the operation. This occurs within the DRC and Extraction functions, and geometrical commands such as **!join** and **!layer**, as well as file reading and writing. If an interrupt is received while drawing to the screen, the drawing immediately terminates, without user confirmation. Script execution is also terminated immediately.

Under Microsoft Windows, pressing the **Pause/Break** key also sends an interrupt signal if *Xic* has the keyboard focus.

When the "wait" cursor is active when the mouse pointer is in a drawing window, *Xic* is "busy". When busy, *Xic* locks out all key press events except for **Ctrl-c**, and most mouse button events. If a locked-out event is received, a pop-up will appear that informs the user that *Xic* is busy and to use **Ctrl-c** to abort the operation. This pop-up will disappear after three seconds (trying to destroy it with the mouse won't work).

When *Xic* is busy and **Ctrl-c** is pressed, the operation may be paused, and the user is asked (on the prompt line) whether to abort or continue. While waiting for input, most buttons are desensitized. Those that are not are the **Help Menu**, **View/Allocation**, and **Attributes/Main Window/Freeze**. Thus, these features are available during the pause.

All other events are dispatched normally while busy, so that visual updates should happen fairly quickly. Unlike early releases, there is no attempt to save unhandled events and handle them later.

Ctrl-e

Pressing **Ctrl-e** prompts the user for a coordinate pair, which is then used in a point operation, just as if the user had clicked with button 1 at that location. When entering coordinates using **Ctrl-e**, the coordinate is not moved to the nearest snap point as it would have been if entered with the mouse. Thus, off-grid points can be entered, and the user must bear this in mind.

When editing a string on the prompt line, **Ctrl-e** will move the cursor to the end of the string.

Ctrl-g

Pressing **Ctrl-g** brings up the **Grid Setup** panel (see 13.11.12). This can be used to alter the grid displayed in the drawing window that had the keyboard focus. This is effectively an accelerator for the **Set Grid** button in the **Main Window** sub-menu of the **Attributes Menu**, or the **Set Grid** button in the **Attributes** menu of sub-windows (see 12.6).

Ctrl-k

When entering text to the prompt area, **trl-K** will delete-to-end. The character over the cursor and all characters to the right will be deleted.

Ctrl-n

The view in a window can be saved at any time by pressing **Ctrl-n**. The view is assigned a letter, which allows it to be recalled with the **View** command. Up to five views can be saved per window, and these are assigned letters A-E in order. The view can also be restored by pressing **Ctrl-Shift-a** through **Ctrl-Shift-e**.

Ctrl-p

In electrical mode, outside of any command, pressing **Ctrl-a** will cause the associated labels of any selected device to also become selected. The associated labels can be deselected by pressing **Ctrl-p**. This is sometimes useful for determining which labels are associated with a given device.

Pressing **Ctrl-p** is equivalent to pressing the **Page Up** key when the DRC **Show Errors** command is active.

Ctrl-r

Pressing **Ctrl-r** will redraw the window which contained the pointer when **Ctrl-r** was pressed.

Ctrl-u

When entering text to the prompt area, pressing **Ctrl-u** will delete all characters from the input buffer.

Ctrl-v

Pressing **Ctrl-v** will bring up a window containing the *Xic* version number and copyright information.

Ctrl-x

Pressing **Ctrl-x** will bring up a the **Expansion Control** panel, the same as the **Expand** command in the **View Menu**.

Ctrl-z

Pressing **Ctrl-z** while the pointer is in a drawing window will iconify *Xic*. **Ctrl-z** in the controlling terminal window retains the usual job control function.

Finally, the **Shift** and **Ctrl** keys are often used in conjunction with the pointer buttons to initiate new operations or modify current operations. The sections describing the commands will provide examples.

3.6 Pointing Device

Xic is most efficiently used with a three-button mouse, trackball, or other input device. The three buttons are normally numbered from the left, with the mouse pointing upward. This manual will refer to buttons by their number according to this convention.

A two-button mouse, as commonly used with PC hardware, does not provide button 2 (the “middle” button). Although a three-button pointing device is recommended, in current *Xic* releases the important button 2 operations can be simulated using button 1 or 3, while holding a modifier key. Thus, for many users, a two-button mouse should be entirely adequate.

In short, button 1 is used for basic point and click operations and menu selections. The middle button, button 2, is used for pan operations in drawing windows, and the right button, button 3, is used for zooming in the drawing windows.

In addition, drawing windows respond to mouse wheel events. The basic action is vertical scrolling, however if **Shift** is held, the window will scroll horizontally. If **Ctrl** is held (which overrides **Shift**) the display will zoom in or out. The mouse wheel sensitivity can be changed with the MouseWheel variable. A mouse wheel will also provide scrolling capability in text windows and the help viewer on most systems.

Button 1 (the left button) is used for point operations in the drawing windows, and for activating command buttons and sliders in menus and pop-ups. In most cases, a "point operation" can be effected in two ways: click twice, or hold and drag. If the pointer does not move too much as button 1 is pressed and released, a single point is defined, and most commands will prompt the user to point a second time to complete the operation. If button 1 is held while the pointer moves, upon release the operation is completed, using the press and release coordinates. A rectangle defining the two positions is typically ghost-drawn while the point operation is in progress.

The delay interval which is used to differentiate a "click" from a "hold" or "drag" can be adjusted by setting the **SelectTime** variable with the **!set** command. The default value is 250 milliseconds, and the adjustable range is 100–1000 milliseconds. Some users may find that setting the delay to a larger value improves the ability to differentiate between the operations described below.

Outside of any command, button 1 performs selection, move/copy, and stretch operations. The **Shift** and **Ctrl** keys act as modifiers for the button 1 presses. The following sections describe the normal operations.

If **Shift**, **Ctrl**, and **Alt** are all held while button 1 is pressed, a "no-operation" (button 4) press is simulated. This performs no action, but updates the coordinate readout window.

If **Shift** and **Ctrl** are both held while clicking on a physical cell instance or electrical subcircuit, a sub-window will appear containing the contents of the subcell or subcircuit. In electrical mode, the new window will display the subcell schematic, and be a proxy for the main window for hypertext, including plot reference points. Clicking in the sub-window will assign hypertext reference points, as if one clicked in the main window (see 3.1.3). This is how one can get hypertext references of assign plot points from a cell that is shown as an instance symbolically.

with only the **Ctrl** key held, clicking on a selected cell instance will provide access to resources as described. If the selected instance is a normal cell, the **Property Editor** panel (see 10.10), with the clicked-on instance as the current object, will appear. If the selected instance master is a parameterized cell (pcell), the **Parameters** panel (see 5.3) appears, allowing the user to reparameterize the instance. If the selected instance is a standard via (see 5.8), the **Standard Via Parameters** panel will appear, allowing the user to alter ther structure of the via.

3.6.1 Basic Selection Operation

If neither of the **Shift** and **Ctrl** keys is pressed, clicking on an object will toggle its selected status. Objects which are selected are drawn with a blinking boundary. These objects are acted on by many of the button commands, so that object selection in an important part of *Xic* operation. The number of selected objects, if any, is displayed in the status area below the layer table. This information is useful, as selected objects can be off-screen, leading to unintended consequences.

The default selection operation is described here. The selection behavior can be modified from the **Selection Control Panel** brought up by the **selcp** button in the top button menu. Only objects on layers that are both visible and selectable (as shown in the layer table) can be selected.

Clicking on a single object will toggle the selection status of the object. If the point where the object was clicked is also over a subcell, the object and not the subcell will be selected or deselected; subcells are affected only if there is no other geometry at the selection point.

It is impossible to select an object or subcell with mouse operations whose boundary is completely invisible in all display windows. Such objects can be deselected, however.

When clicking on an intersecting point of several objects, there are two types of logic available. In the default logic, when clicking on the intersection area of several unselected objects, only one of the objects is selected, and repeatedly clicking in the same spot will selected a different object, deselecting the previous selection if any. Thus, one can cycle through the candidates and select only the one of interest. If two or more of the objects are already selected, only one of the selected objects will be deselected, and no new object will be selected. If exactly one object is selected, it will be deselected, and the "next" object will be selected. If there is no "next" object, then there will be no new selection. The "next" object is subject to the ordering of layers in the layer table (top to bottom) and database ordering (sorted descending in the Y value and ascending in the X value of the upper left corner of the object's bounding box).

In the "legacy" logic, which was used in releases through 2.5.63, clicking on an intersecting point of several unselected objects will select them all. However, clicking on the intersection area of several selected objects will *not* deselect them all. The logic in this case is similar to the default logic. If more that one object is selected, only one of the objects will be deselected per click in an intersecting area. When only one of the objects remains selected, the next click will deselect the selected object, and select the other objects.

If the variable NoAltSelection is set, *Xic* will use the legacy logic.

Clicking (*not* dragging) on an empty part of the drawing will deselect the single object at the head of the selection list, if any, which is the object most recently selected. This applies when no command is active, not when selections are performed within commands.

If neither of the **Shift** or **Ctrl** keys is pressed, and button 1 is pressed, dragged, and released, the selection status of objects that intersect the defined rectangle is toggled. This is an "area select". Unlike clicking (or "point select"), the selection status of all affected objects is toggled by an area select. During the drag, the rectangle defined for the area select is ghost drawn. In area select, qualifying instances are always selected or deselected, whether or not other geometry is present.

A special case applies in both point and area selection, when only physical cell instances are selectable, and three or more instances would be selected. The **Select Instances** pop-up appears, which provides a listing of the selectable instances, along with colored "yes/no" text indicating the present selected state of each instance. The state can be toggled by clicking on the colored text. This is a useful feature for designs containing a large number of overlapping cell instances. The same pop-up may appear in other contexts when instances are being chosen for some operation. In this case, the nomenclature is slightly different ("Choose" instead of "Select"). In both cases, the pop-up is modal, meaning that most interface objects other than the pop-up are locked while the pop-up is visible.

In either point or area select, if the instance bounding box is not visible in the window, the instance will not be selected, which may prevent accidents.

In electrical mode with point selection, objects are acted upon hierarchically. Wires have the highest precedence, followed by labels, instances, and boxes. Only the clicked-on objects with the highest precedence are acted upon, if there are multiple objects clicked on. For example, clicking on a wire over a subcircuit will select or deselect the wire, but ignore the subcircuit. With drag selection, all qualifying objects will be acted upon.

When the selection operation is completed, the status of the modifier keys determines how the chosen objects are processed. If neither of **Shift** or **Ctrl** is pressed, the action is as described. if **Shift** is pressed (but not **Ctrl**), any unselected objects are selected. If **Ctrl** is pressed (but not **Shift**) any selected objects are deselected. If both **Shift** and **Ctrl** are held, the selection status of each object is reversed. This is the default for area selections, but not point selections.

The **desel** button can be used to deselect all selected objects. This acts on all selected objects, whether or not they are on the current layer. The **!select** command is another mechanism whereby objects can be selected.

3.6.2 Basic Move/Copy Operation

Objects must first be selected in order to be moved or copied. These operations are short-cuts to the **Move** and **Copy** commands in the **Edit Menu**. There are also **!mo** (move) and **!co** (copy) commands available for text-mode input from the prompt line.

If the **Shift** key is down when the user presses button 1, and the pointer is over a selected object, then a move/copy operation on all of the selected objects is initiated. Alternatively, pressing button 1 with no keys pressed over a selected object and holding, motionless for a brief period, will similarly initiate a move/copy operation. In the first case, if the user releases button 1 immediately (clicks) then the outlines of the selected objects are "attached" to the pointer and the move/copy operation will complete when the user clicks a second time. Alternatively, the user can drag the pointer (with button 1 still pressed), and the release event will complete the operation. In the second case, the pointer must remain motionless with button 1 down for a brief period. The user can release button 1, at which point the objects are attached to the pointer, and complete the operation with a second button 1 press. Alternatively, the user can begin to drag, and complete the operation by releasing button 1. The brief period of inactivity, or the fact that the **Shift** key is pressed, signals the start of a move/copy operation.

Pressing the **SpaceBar** toggles whether the operation is in move or copy mode. The last state is remembered in the next operation. A message in the prompt area indicates the current mode, which will apply when the operation completes.

When in copy mode, a replication count will be read from the keypress buffer of the current window when the copy is performed. This is an integer, entered by typing into the window. If not found or out of the range 1–100000, a single copy is made. Otherwise, multiple copies will be created, at multiples of the translation distance.

Also in copy mode, when clicking twice rather than dragging, the object being copied remains "attached" to the mouse pointer, so that additional copies can be placed by simply clicking. Pressing **Esc** will terminate this mode.

If the **Shift** key is down when the operation is completed, the angle of translation is constrained to be multiples of 45 degrees. This constraint is visible during the move/copy by observing the behavior or the ghost-drawn outlines as the pointer moves. This is often useful for making sure that the new location is horizontally, vertically, or diagonally aligned with the original location.

If the **Enter** is pressed during a move, when the objects being moved are ghost-drawn and attached to the pointer, the reference point of the object becomes the lower left corner of the bounding box of the objects. Pressing **Enter** will cycle the reference point through the corners of the bounding box, and back to the original reference location. Note that this allows objects that have somehow gotten off-grid to be returned to the grid.

It is possible to change the layer of objects during a move/copy operation. During the time that objects are ghost drawn and attached to the mouse pointer, if the current layer is changed, the objects that are attached can be placed on the new layer. Subcells are not affected.

How this is applied depends on the setting of the LayerChangeMode variable, or equivalently the settings of the **Layer Change Mode** pop-up from the **Set Layer Chg Mode** button in the **Modify Menu**. The possible actions are to ignore the layer change, place objects originating from the old current layer on the new layer, or to place all new objects on the new layer. If the current layer is set back to the previous layer before clicking to locate the new objects, no layers will change. Note that layer change is only possible for "click-click" mode and not "press-drag".

3.6.3 Basic Stretch Operation

Objects must first be selected in order to be stretched. The basic stretch operation described here is also available from the **Stretch** command in the **Edit Menu**, but that command provides additional features, such as vertex selection, not available from the basic operation. Stretching operations are also available for polygons in the **polyg** command, and for wires in the **wire** command.

Clicking on a selected object with the **Ctrl** key pressed initiates a stretch. If the **Shift** key is also held, an actual stretch command is initialted, as if the **Stretch** button in the **Modify** menu was pressed. The mode changes to the stretch command, which can be terminated by pressing the **Esc** key. The command allows use of vertex selection to mark and move several polygon vertices in tandem, a feature not available in the simple stretch operation to be described, which in initiated if the **Shift** key is not also pressed.

Any object other than subcells can be stretched, but the effect of the stretch differs on the various objects. Boxes and labels are stretched in such a way as to maintain a rectangular shape. That is, if a corner is stretched, the adjacent vertices are also moved in order to keep the internal angles 90 degrees.

The stretch operation works differently on Manhattan polygons than polygons containing nonorthogonal angles. For non-Manhattan polygons, a single vertex is moved, all others remain fixed. The stretch operation on Manhattan polygons is similar to the operation as applied to boxes, i.e., the corner and adjacent vertices are changed so as to keep the polygon Manhattan. A single vertex can be stretched arbitrarily either by selecting the vertex in the **Stretch** command in the **Edit Menu**, or by using the vertex editor in the **polyg** command.

If the **Ctrl** key is pressed when the user presses button 1, and the pointer is over a selected object that is not a subcell, a stretch operation will be initiated. The operation is performed on all selected objects, and the new outlines are ghost drawn. As for move/copy, the operation can be performed by clicking twice, or by dragging and releasing button 1. For selected polygons and wires, the vertex nearest the button 1 press location, for each object, is moved. For boxes and labels, the corner closest to the button down location is moved.

If the **Shift** key is pressed when the stretch is completed, the angle of translation is constrained to multiples of 45 degrees. This can be seen in the behavior of the ghost drawn outlines while the pointer moves, with and without the **Shift** key pressed. At this stage, the **Ctrl** key is ignored.

3.6.4 Additional Notes

Pressing the **Esc** key will terminate the operations described above while in progress. The **Tab** and **Shift-Tab** keys will undo and redo the operation, respectively. These operations sound complex when described in print, but become quite natural in practice. The user should spend a few minutes learning these operations.

In the layer menu, button 1 selects the current layer, as indicated by the highlight box drawn around the entry. If the **Shift** or **Ctrl** key is pressed while clicking with button 1 in the layer menu, the action is identical to a button 2 press, i.e., the layer visibility status is changed. This is advantageous for users with a two-button pointing device, on which button 2 is usually absent.

Many of the pop-up windows can be moved by pressing button 1 while the pointer in on the background or a label object in the pop-up. While button 1 is held, the outline of the pop-up is ghost-drawn and attached to the pointer. The pop-up is moved to the new location when button 1 is released.

3.6.5 Button 2 Operations

Button 2 a is usually the center button on a three-button pointing device. On two-button mice, the right button is typically button 3, and button 2 is missing. On some systems, pressing buttons 1 and 3 simultaneously will simulate a button 2 press. *Xic* provides alternative ways to perform the button 2 operations, so that a two-button pointing device can be used, but is a tiny bit less efficient.

If button 2 is clicked in a drawing window, the window is redrawn with the click location centered in the window. If instead button 2 is pressed and the pointer moved to a new location before release, the window is redrawn with the press location moved to the release location. If there are multiple windows open, only the window under the release will be redrawn. Thus, for example to change the view in a sub-window, press and hold button 2 while pointing at the desired feature in the main (or another) window, then release button 2 while pointing in the sub-window. The sub-window will show the pointed-to objects at the release location.

The same action will be initiated if button 3 is pressed while either the **Shift** or **Ctrl** key is held down. The key state when button 3 is released does not matter.

In the layer menu, button 2 will switch the visibility of layers, as indicated by the sample box. Clicking button 2 on the individual layers toggles their visibility. Clicking button 2 on the small box icon at the far right of the layer menu will toggle visibility of all layers. All layers will be set to visible or invisible according to whether a majority of layers were originally invisible or visible, respectively.

The behavior is a little different between physical and electrical modes. In physical mode, the screen will not be redrawn automatically, unless the **Shift** key is held during the button 2 press, but can be redrawn by clicking button 2 in the center of the drawing window, or by pressing the **Ctrl-r** key combination.

In electrical mode, the screen is automatically redrawn. The SCED (drawing) layer is always visible. Instead of the visibility of this layer being toggled, the fill setting is toggled between solid and empty fill.

The same behavior is obtained by holding **Shift** or **Ctrl** while clicking with button 1 in the layer menu. If **Shift** is held, the screen will be redrawn automatically while in physical mode.

3.6.6 Button 3 Operations

Button 3 performs a zoom operation. Draagging or clicking twice defines diagonal corners of a rectangle to zoom into. The window will then display the contents of this area (after compensating for aspect ratio).

If the same operation is done, but **Ctrl** or **Shift** is pressed during the drag button-up or the second mouse click, operation is different. In this case, the area is marked by a dotted highlighting box, and a subsequent button 3 press will complete the operation. A press in the same window will cause the area defined by the first and second points to be shrunk by the ratio of the diagonals of the rectangles defined by point 1, point 2 and point 1, point 3. To zoom in a lot, point 2 is much closer to point 1 than point 3 is to point 1. Alternatively, a button 3 press in a different window will display the boxed area of the first window in the second window.

If **Shift** or **Ctrl** is held down before the initial button 3 press in a drawing window, a pan operation will be initiated instead of the zoom, the same as if button 2 was pressed.

In the layer menu, button 3 enables layer blinking, if neither of **Shift** or **Ctrl** is pressed. Pressing and holding button 3 over a layer entry in the layer table will cause that layer to blink periodically in the drawing windows, while button 3 remains pressed. Layers that happen to have the same color as the selected blinking layer will also blink, since the operation is sensitive only to the layer color.

In combination with **Shift** and **Ctrl**, clicking with button 3 on a layer entry provides a shortcut:

- **Ctrl**-button 3 will set the current layer to the clicked-on layer, and bring up the **Color Selection** panel, loaded with that layer's color.
- **Shift**-button 3 will set the current layer to the clicked-on layer, and bring up the **Fill Pattern Editor** loaded with that layers pattern.
- **Ctrl**-**Shift**-button 3 will set the current layer to the clicked-on layer, and bring up the **Tech Parameter Editor** targeted to the layer.

3.6.7 Button 4

Support is provided for a fourth button for those pointing devices which have four buttons. Pressing button 4 does nothing except update the coordinates displayed on-screen. No action is performed. This can be simulated by holding the **Ctrl**, **Shift**, and **Alt** keys while pressing button 1.

3.6.8 Mouse Wheel

The GTK user interface provides support for mouse wheels. Any window that has scroll bars can be scrolled by moving the pointer *over a scroll bar* and turning the mouse wheel. The drawing windows, most text windows and help viewer windows respond to the mouse wheel by scrolling when the pointer is in the window, as well as over a scroll bar (if any). In drawing windows, scrolling will be horizontal if **Shift** is held, and if **Ctrl** is held (which overrides **Shift**), the display will zoom in or out instead. The mouse wheel sensitivity can be changed with the MouseWheel variable.

3.7 The WR Button: Email Client

Keyword: `mail`

The **WR** button is located in the upper left corner of the *Xic* main window. Pressing this button brings up a mail client window. The mail client can be used to send mail to any email address, though when the panel appears, it is pre-loaded with the address of Whiteley Research technical support. The text field containing the address, as well as the subject, can be changed.

The mail client can be used to send mail to arbitrary mail addresses, though when the panel appears, it is pre-loaded with the address of Whiteley Research technical support. The text field containing the address, as well as the subject, can be changed.

The main text window is a text editor with operations similar to the text editor used elsewhere in *Xic* and *WRspice*. The **File** menu contains commands to read another text file into the editor at the location of the cursor (**Read**), save the text to a file (**Save As**) and send the text to a printer (**Print**). When done, the **Send Mail** command in the **File** menu is invoked to actually send the message. Alternatively, one can quit the mail client without sending mail by pressing **Quit**.

The **Edit** menu contains commands to cut, copy, and paste text.

The **Options** menu contains a **Search** command to find a text string in the text. The **Attach** command is used to add a mime attachment to the message. Pressing this button will cause prompting for the name of a file to attach. While the prompt pop-up is visible, dragging a file into the mail client will load that file name into the pop-up. This is also true of the **Read** command. Attachments are shown as icons arrayed along the tool bar of the mail client. Pressing the mouse button over an attachment icon will allow the attachment to be removed.

In the Windows version, since Windows does not provide a reliable interface for internet mail, the mail client and crash-dump report may not work. Mail is sent by passing the message to a Windows interface called "MAPI", which in turn relies on another installed program to actually send the mail. In the past, the mail system was known to work if Outlook Express was installed and configured as the "Simple MAPI mail client". It is unknown whether this is still an option with recent Windows releases.

To get mail working in Windows 8, it was necessary to download and install something called "`live mail`" from Microsoft, which eventually worked. This application supports MAPI, apparently the default Windows 8 Mail application does not. The default Windows 8 Mail application also does not work with POP3 servers.

3.8 Top Button Menu

The top button menu extends along the top of the *Xic* main window, just below the main menu bar. This contains a number of buttons and other controls. In left-to-right order, these are described briefly below, and in more detail in the sections that follow.

The **lsearch** button and entry: find layer and set current
: The text entry displays the name of the current layer. This entry area and the adjacent button with the blue triangle icon can perform a layer search by (partial) name. Matching layers become the current layer.

The **ltvis** button: show/hide layer table
: This button toggles visibility of the layer table.

The **lpal** button: show/hide layer palette
: This button controls visibility of the layer palette.

The **setcl** button: set current layer form clicked-on object
: Pressing this button, then clicking on an object in a drawing window will set the current layer to the layer of the object.

The **selcp** button: show/hide selection control panel
: This button controls the visibility of the **Selection Control** panel.

The **desel** button: deselect all objects
Pressing this button will deselect all currently selected objects.

The **rdraw** button: redraw windows
Pressing this button will redraw the main window, and all sub-windows showing the same display mode (electrical or physical) as the main window.

The coordinates readout
This window displays the coordinates of the mouse pointer.

3.8.1 The lsrch Button and Entry: Find Layer and Set Current

Keyword: `lsrch`

Just above the layer table, at the far left of the top button menu, is a text entry area, with a button containing a blue triangle icon to the left. The name of the current layer is displayed in this area. This can be used to find layers by name. One can enter the first few characters of a layer name into the text area, then press the button to the left. The button icon will change to two triangles, and the layer table will scroll to the first matching layer found (if any), as the current layer. Clicking the button a second and subsequent time will scroll to the next and later matches. Though the text in the entry area will take on the selected layer name, the search string is retained internally as long as the two-triangle icon is displayed on the button. This will revert to the single triangle after a few seconds if not clicked. When using the *layer*`:`*purpose* form, both the layer and purpose strings are handled independently, and both can contain just the first few characters of the actual layer and purpose names.

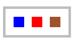
3.8.2 The ltvis Button: Show/Hide Layer Table

Keyword: `ltvis` 

The **ltvis** button in the top button menu toggles display of the layer table. As the layer table occupies significant screen area, it is sometimes useful to get rid of it to enable a larger main drawing window.

Much of the functionality of the layer table is found in the layer palette which in some ways is like a "mini layer table" containing only a few chosen layers. Even without the palette, one can switch the current layer using the layer search capability, or the **setcl** button, both found in the top button menu.

3.8.3 The lpal Button: Show/Hide Layer Palette

Keyword: `lpal` 

The **lpal** button in the top button menu will bring up the layer palette. The layer palette is an adjunct to the layer table which provides a means for quick access to a few "important" layers, and prints information about layers. This is particularly useful when working with technologies containing a large number of layers, to avoid hunting through the layer table. When the mouse pointer hovers over a layer indicator in the layer table or in the palette, information about that layer is printed in the top part of the palette.

The layer palette consists of three logical sections. The top section is a text area that displays information about the layer currently or was last under the mouse pointer. The user can move the pointer over the layer icons in the layer table or the palette, and the palette will display the information. The information printed includes the alias and description of the layer, and the GDSII mapping layer/datatype numbers.

In the lower section, there are four rows of locations for layer indicators. The indicators in this section can be dragged and clicked on in the same manner and same functionality as layers in the layer table. The top row contains layer indicators for the last five choices of current layer. This row is automatically updated whenever the user selects a current layer by any means.

The three rows below can be filled by the user, by dragging/dropping layers from the layer table, or from the top row in the palette. Layers in these rows can be dragged/dropped within the rows to change the listing order. A layer indicator can be removed from these rows by pressing the **Remove** button at the top of the panel, then clicking on a layer indicator in this area. The indicator will disappear, and the **Remove** button will become unselected.

In order to conserve space, only the index number of the layer in the layer table is shown with the layer sample box in the layer palette. The layer's name and other information can be obtained by hovering over the indicator with the mouse pointer.

The palette layers can be saved in one of seven registers and restored later, with the **Save** and **Restore** buttons. There are separate registers for physical and electrical modes, so that the same register number can be used in each mode. The current palette is saved when the palette is dismissed, and restored when the palette is popped up again.

These registers are saved in a technology file created with the **Save Tech** button in the **Attributes Menu**. The corresponding technology file keywords are `PhysLayerPalette1` – `PhysLayerPalette7` and `ElecLayerPalette1` – `ElecLayerPalette7`. Each keyword can be set to a space-separated list of layer names, representing the content and order of the layers in the register.

3.8.4 The setcl Button: Set Current Layer from Clicked-On Object

Keyword: `setcl` 

The **setcl** button in the top button menu allows setting the current layer by clicking on objects in a drawing window. The user must first press the **setcl** button, then click on an object in a drawing window. The current layer will be reset to the layer of that object. Without changing the mouse pointer location, clicking will cycle through other layers of objects that were under the original click location. Additional clicks must come within a short period of time, or the command will exit first.

3.8.5 The selcp Button: Show/Hide Selection Control Panel

Keyword: `selcp` 

The **selcp** button in the top button menu displays the **Selection Control Panel** which provides a number of mode switches which control object selection.

There are three "radio button" groups. The **Pointer Mode** group sets the mode for selections initiated with button 1 while outside of commands. There are three choices:

Normal
: Standard select/modify behavior.

Select
Allow selections only.

Modify
Allow move/copy/stretch on selected objects only.

The **Area Mode** group provides three modes for area (drag-over) selections.

Normal
Standard area selection behavior, objects are chosen if the object touches but does not completely cover the selection area.

Enclosed
Chosen objects must exist completely within the selection area.

All
Any object that touches the selection box is chosen.

The **Selections** group modifies how chosen objects are processed.

Normal
Standard behavior.

Toggle
Reverse the selected/deselected status of all chosen objects.

Add
Select all unselected objects chosen.

Remove
Deselect all selected objects chosen.

While selecting, and the **Selections** group is Normal, during completion of the selection operation, the modifier keys are recognized:

Shift
Select all unselected objects chosen.

Ctrl
Deselect all selected objects chosen.

Shift-Ctrl
Reverse the selected/deselected status of all objects chosen.

Thus, the Toggle/Add/Remove modes can be established transiently with the modifier keys. For area selection, the normal operation is to toggle the selections. For a point select (mouse click), if more than one underlying object is selected, one of the selected objects is deselected, and there is no new selection.

The **Objects** group specifies the type of objects that can be selected and deselected with mouse operations. The buttons are labeled **Cells**, **Boxes**, **Polys**, **Wires**, and **Labels**. These buttons control whether or not the indicated type of object can be selected or deselected with the mouse. This is useful, for example, when one needs to select cells that are covered by geometric objects, since the geometric objects will always be selected with a mouse click, and not the cells.

Normally, when scanning through the database for objects that are within the selection area, layers are searched from logical top to bottom. The logical top layer is the last layer listed in the layer table (i.e., at the bottom). Thus, in some modes objects on upper layers will be selected preferentially over objects on lower layers. If the **Search Up** button is active, this ordering is reversed, layers are searched from logical bottom to top, or top to bottom as listed in the layer table.

In the extraction system, the search order will affect the default association of terminals to layers. It also applies to the operations in the **Path Selection Control** panel.

3.8.6 The desel button: Deselect Objects

Keyword: `desel`

Pressing the **desel** button will deselect all of the currently selected objects. Individual or groups of objects can be deselected by selecting them a second time with the mouse. When not in a command mode, pressing the **Esc** key will also deselect all selected objects.

3.8.7 The rdraw button: Redraw Windows

Keyword: `rdraw`

Pressing this button will redraw the main window, and any sub-windows that are showing the same display mode (electrical or physical). The drawing window with keyboard focus can also be redrawn by typing **Ctrl-r**. Clicking with button 2 near the center of the window is yet another way to force a redraw. After most operations, the windows are automatically redrawn, so forcing a redraw is not often needed. Exceptions are when changing layer colors and fill patterns.

3.8.8 Coordinates Display

Just above the *Xic* main drawing window is an area where pointer coordinates are printed. The coordinates are given in microns, relative to the internal coordinate system. In physical mode, the origin is indicated on-screen. The first row in the coordinate display is the current location of the pointer. The second row is the location of the last button press event. The third row is the delta between the current position and the last button press event.

3.9 Main Drawing Window

The main drawing window occupies the largest section of the visible user interface. This is the primary presentation and work area for editing. The main drawing window supports drag and drop as a drop receiver for files.

Drawing windows respond to a number of button operations and key presses to pan and zoom. See the sections on button and key operations for a complete description. In addition, drawing windows respond to mouse wheel events. The basic action is vertical scrolling, however if **Shift** is held, the window will scroll horizontally. If **Ctrl** is held (which overrides **Shift**) the display will zoom in or out. The mouse wheel sensitivity can be changed with the MouseWheel variable.

Xic supports standard drag and drop protocols. One is able to drag files from many file manager programs into the main window of *Xic*, and that file will be loaded into *Xic*. The **File Selection** panel from the **File Select** button in the **File Menu**, and the **Files Listing** pop-up from the **Files List** button in the **File Menu**, participate in the protocols as sources and receivers. The text editor and mail client pop-ups, among others, are drop receivers. While in text editing mode, the prompt line is a drop receiver, and drops in the main window are redirected to the prompt line when editing mode is active. Most of the pop-ups in *Xic* which solicit a text string are also drop receivers.

The file must be a standard file on the same machine. If it is from a tar file, or on a different machine, first drag it to the desktop or to a directory, then into *Xic*. The GNOME gmc file manager allows one to view the contents of tar files, etc. as a "virtual file system". Window Maker and Enlightenment window managers, at least, are drag/drop aware.

Most of the listing pop-ups in *Xic* are drag sources, i.e., one can drag the name from the listing and drop it in a drawing window.

When a window is displaying cells from a Cell Hierarchy Digest (CHD), meaning that the **Display** button in the **Cell Hierarchy Digests** panel is engaged, the dropped cell name must match a cell name in the CHD. If not, an error message will appear. Otherwise, the display will switch to the dropped cell as the root. Changing the display root does *not* change the default cell of the CHD. In this mode, nothing new is brought into program memory.

In normal display mode, the window will open the cell or file dropped. The dropped object can be of various types, depending on the source: file names, cell names from memory, cell names from a CHD, and library references are all possible. If the dropped object does not suggest an unambiguous cell, a pop-up will appear requesting that the user make a selection from a given listing. This may happen, for example, when a dropped file name contains more than one top-level cell, or the dropped name is a library containing multiple references.

A dropped file name will cause the file to be read into memory, and the top-level cell will be displayed. A cell name from a CHD will cause the cell and its hierarchy to be extracted from the CHD's source and loaded into memory, and the given cell will be displayed. Library references that point to a cell will likewise be brought into memory, and the referenced cell will be displayed. A cell name will simply display that cell, which if not already in memory, will be opened through the library and search path mechanism, or created internally as an empty cell if unresolved.

If dropped into the main drawing window, the displayed cell becomes the current cell for editing and selections. If dropped in a sub-window, the cell will be displayed, but can not be edited if it is different from the current cell (the cell shown in the main drawing window).

3.10 *Xic* Layers

In *Xic*, boxes, polygons, and other objects are created on *layers*. These often correspond to mask levels in a fabrication process, but the actual interpretation is up to the user.

Most often, layers are defined in the technology file, and these are shown within *Xic* in the layer table. One of the layers is selected as the "current layer", which is used for drawing objects.

Layers have an order, as shown in the layer table display. Layers that come later in the listing are considered to be "above" the layers listed earlier. This is reflected in how layouts are drawn on-screen and in plots, as the fill (if any) of a layer will obscure the lower layers.

Historically, *Xic* has used a very simple model for layers based on CIF. In this model, each layer has a unique name of four characters or fewer.

Starting with the *Xic*-3.3 branch, the OpenAccess model is used. This provides fundamental compatibility with design tools based on the OpenAccess database, including Cadence Virtuoso. However, it is a bit more complicated.

The word "layer" now has two meanings. This is unfortunate, but the meaning should be clear in context. First, there are the *Xic* layers we have mentioned. Second, there is the concept of a component (or OpenAccess) layer. In OpenAccess, layer names are associated with layer numbers, forming an abstraction that can be identified by name or number. OpenAccess also similarly defines another abstract type called the "purpose". Again, there are purpose names and purpose numbers, and an abstraction identifiable by name or number. In order to draw an object in OpenAccess, one requires a layer and a purpose. A layer and a purpose in OpenAccess is called a layer/purpose pair (LPP). An LPP is actually what corresponds to an *Xic* layer.

In *Xic*, there is a default purpose, with name "`drawing`". When a purpose name is not explicitly specified, this purpose will be assumed.

Every *Xic* layer has a component layer name and purpose. The name of an *Xic* layer is given or printed in the form

component_layer[`:`*purpose*]

If the purpose name is "`drawing`", then it is not printed or given explicitly. Otherwise, the purpose is separated from the component layer name by a colon ('`:`') character. Note that when the purpose is "`drawing`", the *Xic* layer name is simply the component layer name, so if the only purpose used is "`drawing`", the distinction between OpenAccess and *Xic* layer names vanishes.

Example *Xic* layer names:

```
m1
m1:pin
```

The first name corresponds to component layer name `m1` and purpose `drawing`. The second example uses a purpose named "`pin`".

In *Xic*, layer names of both types, and purpose names, are always recognized and treated without case-sensitivity. There is no limit on the length of these names. Component layer and purpose names can contain alphanumeric characters plus dollar sign ('`$`') and underscore ('`_`').

All of the component layer and purpose names also have corresponding numbers. These may be assigned by the user, or assigned internally by *Xic*. *Xic* will maintain the associations, but the numbers are not used by *Xic*. They are, however, important for compatibility with other tools.

All *Xic* layers may be given an alias name. The layer will be recognized by this name, as well as its normal name. *Xic* layers may also contain a description string, presentation attributes such as color and fill pattern, and a host of other flags and properties for use within *Xic*.

3.11 Layer Table

The layer table is arrayed vertically to the left of the main drawing window. If layers have been specified to *Xic*, they will be shown in this area. If there are more layers than space available for display, a scroll bar is provided. There is no limit on the number of layers that can be defined in *Xic*. Separate layer tables are provided for electrical and physical modes.

The "grip" that separates the layer table from the main drawing window can be dragged to change the layer table width.

To the left of each entry sample box are indicators that when clicked on will toggle either the visibility or selectability of that layer. If the layer is not visible, objects on that layer will not be shown in layout images. It the layer is not selectable, objects on the layer can't be selected.

To the right of the sample box are the layer name and purpose names.

When the layer is not visible, the sample box is not drawn, and the green "`v`" indicator becomes a red "`nv`". Layers with the `Invisible` technology file keyword will by default be invisible. If the layer is not selectable, the layer name / purpose name area is shown with a dark background, and the green "`s`" indicator becomes a red "`ns`". Layers with the `NoSelect` technology file keyword will by default be non-selectable.

Visibility can be toggled by clicking on the `v/nv` indicator with button 1, or by clicking in the sample box area with button 2, or by clicking anywhere in the entry with button 1 and the **Shift** key held.

In releases earlier than 4.1.6, a layer visibility change would not automatically redraw the screen in physical mode. This is ancient behavior intended to accommodate slow screen redraws. When several layer visibility changes are to be made, one can make the changes and then force a screen redraw. This seems to be unnecessary on newer computers, which render very quickly, so the updating is now automatic. There is a variable, NoPhysRedraw, that if set will revert to the original behavior of no automatic redraw in physical mode, if the user prefers this.

Pressing **Shift** along with clicking button 2 in the sample box area will suppress redraw if the variable is not set. If the variable is set, then the **Shift**-click will redraw tha main window and all similar sub-windows after the operation. The drawing window that has the keyboard focus can be redrawn by pressing **Ctrl-r**. The **rdraw** button to the left of the coordinate readout will redraw the main window and all similar sub-windows.

In electrical mode, the SCED layer, which is the electrical mode active wiring layer, is always visible. Instead, of toggling visibility of this layer, the button presses will toggle between solid and empty fill.

Selectability can be toggled by clicking on the `s/ns` indicator with button 1, or by clicking in the layer name/purpose name area with button 2, or by clicking anywhere on the entry with button 1 and the **Ctrl** key held.

One can also toggle the visibility and selectability states of all layers except for the current layer. At the bottom of the layer table, there are two gray areas labeled "`vis`" and "`sel`".

Clicking the "`vis`" area with button 1 or button 2 will switch all layers except for the current layer to invisible, and back. The comment above regarding window redraw in physical mode applies here as well. If **Shift** is held while clicking, the current redrawing behavior is reversed. When switching back to "all layers visible", layers with the `Invisible` keyword applied in the technology file will remain invisible.

Similarly, clicking the "`sel`" area will switch all layers except for the current layer to non-selectable and back. When switching back to "all layers selectable", layers with the `NoSelect` keyword applied in the technology file will remain non-selectable.

Button 3 enables layer blinking, if neither of **Shift** or **Ctrl** is pressed. Pressing and holding button 3 over a layer entry in the layer table will cause that layer to blink periodically in the drawing windows, while button 3 remains pressed. Layers that happen to have the same color as the selected blinking layer will also blink, since the operation is sensitive only to the layer color.

In combination with **Shift** and **Ctrl**, clicking with button 3 on a layer entry provides a shortcut:

- **Ctrl**-button 3 will set the current layer to the clicked-on layer, and bring up the **Color Selection** panel, loaded with that layer's color.
- **Shift**-button 3 will set the current layer to the clicked-on layer, and bring up the **Fill Pattern Editor** loaded with that layers pattern.
- **Ctrl-Shift**-button 3 will set the current layer to the clicked-on layer, and bring up the **Tech Parameter Editor** targeted to the layer.

The current layer is shown with a blue highlighting box. Clicking on a layer entry with button 1 will make it the current layer. The current layer is used when creating objects in the layout.

One can also search for a layer to set as the current layer by name. Just above the layer table is a text entry area, with a button containing a blue triangle icon to the left. The name of the current layer is displayed in this area. This can be used to find layers by name. One can enter the first few characters of a layer name into the text area, then press the button to the left. The button icon will change to two triangles, and the layer table will scroll to the first matching layer found (if any), as the current layer. Clicking the button a second and subsequent time will scroll to the next and later matches. Though the text in the entry area will take on the selected layer name, the search string is retained internally as long as the two-triangle icon is displayed on the button. This will revert to the single triangle after a few seconds if not clicked. When using the *layer*:*purpose* form, both the layer and purpose strings are handled independently, and both can contain just the first few characters of the actual layer and purpose names.

The current layer can also be set with the **setcl** button in the top button menu. If one presses this button, then clicks on an object in a drawing window (the object must be contained in the current cell), the current layer will be changed to the object's layer. All of the rules for selections apply when interpreting which object will specify the layer, and in particular the object must be selectable.

The **ltvis** button in the top button menu will toggle the visibility of the layer table. The layer table takes a lot of screen area, and often it is not needed. The layer palette can be used instead to provide access to a few chosen layers.

3.12 Status Display

The status area is located below the prompt line. This area provides information about current program modes. It displays the technology name from the technology file, if any, the current cell name, the grid spacing, the snap number if not 1, the number of objects selected if any, and the level of subedit in a **Push**, if in a subedit. Also displayed is a mode keyword, or "`MAIN`", and a code representing the current transform if set. If the current cell has been modified and not saved to disk, "`Mod`" will appear in the status area in colored text. If the current cell has the IMMUTABLE flag set, "`RO`" (for "read only") will appear. If the physical grid origin is not 0,0 (set with the PhysGridOrigin variable), "`PhGridOffs`" will be displayed in colored text.

Dragging over text in the status display with button 1 held down will select the text. Clicking on a word with button 1 will select the word. Selected text is available for export to other windows, as the primary selection in Unix/Linux, or from the clipboard in Windows. Under Windows, the selection is copied to the Windows clipboard automatically.

3.13 Text Entry Windows

The GTK interface provides single and multi-line text entry windows for use in the graphical interface. These entry areas use a common set of key bindings (see 3.13.4) and respond to and use the system clipboard (see 3.13.3) and other selection mechanisms in the same way.

3.13.1 Single-Line Text Entry

In many operations, text is entered by the user into single-line text-entry areas that appear in pop-up windows. These entry areas provide a number of editing and interprocess communication features which are described with the key bindings (see 3.13.4) and system clipboard (see 3.13.3).

In both Unix/Linux and Windows, the single-line entry is typically also a receiver of drop events, meaning that text can be dragged form a drag source, such as the **File Manager**, and dropped in the entry area by releasing button 1. The dragged text will be inserted into the text in the entry area, either at the cursor or at the drop location, depending on the implementation.

3.13.2 The Text Editor

The graphical interface provides a general-purpose text editor window. It is used for editing text files or blocks, and may be invoked in read-only mode for use as a file viewer. In that mode, commands which modify the text are not available.

This is not the world's greatest text editor, but it works fine for quick changes and as a file viewer. For industrial-strength editing, a favorite stand-alone text editor is probably a better choice.

The following commands are found in the **File** menu of the editor. Not all of these commands may be available, for example the **Open** button is absent when editing text blocks.

Open
Bring up the **File Selection** panel. This may be used to select a file to load into the editor. This is the same file manager available from the **File Select** button in the *Xic* **File Menu**.

Load
Bring up a dialog which solicits the name of a file to edit. If the current document is modified and not saved, a warning will be issued, and the file will not be loaded. Pressing **Load** a second time will load the new file, discarding the current document.

Read
Bring up a dialog which solicits the name of a file whose text is to be inserted into the document at the cursor position.

Save
Save the document to disk, or back to the application if editing a text block under the control of some command.

Save As
Pop up a dialog which solicits a new file name to save the current document under. If there is selected text, the selected text will be saved, not the entire document.

Print
Bring up a pop-up which enables the document to be printed to a printer, or saved to a file.

Write CRLF
This menu item appears only in the Windows version. It controls the line termination format used in files written by the text editor. The default is to use the archaic Windows two-byte (DOS) termination. If this button is unset, the more modern and efficient Unix-style termination is used. Older Windows programs such as Notepad require two-byte termination. Most newer objects and programs can use either format, as can the *XicTools* programs.

Quit
Exit the editor. If the document is modified and not saved, a warning is issued, and the editor is not exited. Pressing **Quit** again will exit the editor without saving.

The editor can also be dismissed with the window manager "dismiss window" function, which may be an '**X**' button in the title bar. This has the same effect as the **Quit** button.

The editor is sensitive as a drop receiver. If a file is dragged into the editor and dropped, and neither of the **Load** or **Read** dialogs is visible, the **Load** dialog will appear with the name of the dropped file preloaded into the dialog text area. If the drop occurs with the **Load** dialog visible, the dropped file name will be entered into the **Load** dialog. Otherwise, if the **Read** dialog is visible, the text will be inserted into that dialog.

If the **Ctrl** key is held during the drop, and the text is not read-only, the text will instead be inserted into the document at the insertion point.

The following commands are found in the **Edit** menu of the text editor.

Undo This will undo the last modification, progressively. The number of operations that can be undone is unlimited.

Redo This will redo previously undone operations, progressively.

The remaining entries allow copying of selected text to and from other windows. These work with the clipboard provided by the operating system, which is a means of transferring a data item between windows on the desktop (see 3.13.3).

Cut to Clipboard
Delete selected text to the clipboard. The accelerator **Ctrl-x** also performs this operation. This function is not available if the text is read-only.

Copy to Clipboard
Copy selected text to the clipboard. The accelerator **Ctrl-c** also performs this operation. This function is available whether or not the text is read-only.

Paste from Clipboard
Paste the contents of the clipboard into the document at the cursor location. The accelerator **Ctrl-v** also performs this operation. This function is not available if the text is read-only.

Paste Primary (Unix/Linux only)
Paste the contents of the primary selection register into the document at the cursor location. The accelerator **Alt-p** also performs this operation. This function is not available if the text is read-only.

The following commands are found in the **Options** menu of the editor.

Search

Pop up a dialog which solicits a regular expression to search for in the document. The up and down arrow buttons will perform the search, in the direction of the arrows. If the **No Case** button is active, case will be ignored in the search. The next matching text in the document will be highlighted. If there is no match, "not found" will be displayed in the message area of the pop-up.

The search starts at the current text insertion point (the location of the I-beam cursor). This may not be visible if the text is read-only, but the location can be set by clicking with button 1. The search does not wrap.

Font

This brings up a tool for selecting the font to use in the text window. Selecting a font will change the present font, and will set the default font for new text editor class windows. This includes the file browser and mail client pop-ups.

The GTK interface provides a number of default key bindings (see 3.13.4) which also apply to single-line text entry windows. These are actually programmable, and the advanced user may wish to augment the default set locally.

3.13.3 Selections and Clipboards

Under Unix/Linux, there are two similar data transfer registers: the "primary selection", and the "clipboard". both correspond to system-wide registers, which can accommodate one data item (usually a text string) each. When text is selected in any window, usually by dragging over the text with button 1 held down, that text is automatically copied into the primary selection register. The primary selection can be "pasted" into other windows that are accepting text entry.

The clipboard, on the other hand, is generally set and used only by the GTK text-entry widgets. This includes the single-line entry used in many places, and the multi-line text window used in the text editor (see 3.13.2), file browser, and some other places including error reporting and info windows. From these windows, there are key bindings that cut (erase) or copy selected text to the clipboard, or paste clipboard text into the window. The cut/paste functions are only available if text in the window is editable, copy is always available.

Under Windows there is a single "Windows clipboard" which is a system-wide data-transfer register that can accommodate a single data item (usually a string). This can be used to pass data between windows. In use, the Windows clipboard is somewhat like the Unix/Linux clipboard.

Text in most text display windows can be selected by dragging with button 1 held down, however the selected text is not automatically added to the Windows clipboard. On must initiate a **cut** or **copy** operation in the window to actually save the selected text to the Windows clipboard. The "copy to clipboard" accelerator **Ctrl-c** is available from most windows that present highlighted or selected text. Note that there is no indication when text is copied to the clipboard, the selected text in all windows is unaffected, i.e., it won't change color or disappear. The user must remember which text was most recently copied to the Windows clipboard.

Clicking with button 2 will paste the primary selection into the line at the click location, if the window text is editable.

Clicking with button 3 will will bring up a context menu. From the menu, the user can select editing operations.

The GTK interface hides the details of the underlying selection mechanisms, creating a consistent interface under Windows or Uniix/Linux. There is one important difference, however: in Windows, the primary selection applies only to the program containing the selection. In Unix/Linux, the primary selection applies to the entire desktop.

3.13.4 GTK Text Input Key Bindings

The following table provides the key bindings for editable text entry areas in GTK-2. However, be advised that these bindings are programmable, and may be augmented or changed by installation of a local theme.

GTK Single-Line Bindings

Ctrl-a	Select all text
Ctrl-c	Copy selected text to clipboard
Ctrl-v	Paste clipboard at cursor
Ctrl-x	Cut selection to clipboard
Home	Move cursor to beginning of line
End	Move cursor to end of line
Left	Move cursor left one character
Ctrl-Left	Move cursor left one word
Right	Move cursor right one character
Ctrl-Right	Move cursor right one word
Backspace	Delete previous character
Ctrl-Backspace	Delete previous word
Clear	Delete current line
Shift-Insert	Paste clipboard at cursor
Ctrl-Insert	Copy selected text to clipboard
Delete	Delete next character
Shift-Delete	Cut selected text to clipboard
Ctrl-Delete	Delete next word

Clicking with button 1 will move the cursor to that location. Double clicking will select the clicked-on word. Triple clicking will select the entire line. Button 1 is also used to select text by dragging the pointer over the text to select.

Clicking with button 2 will paste the primary selection into the line at the click location, if the window text is editable.

Clicking with button 3 will will bring up a context menu. From the menu, the user can select editing operations.

These operations are basically the same in Windows and Unix/Linux, with one important difference: in Windows, the primary selection applies only to the program containing the selection. In Unix/Linux, the primary selection applies to the entire desktop, like the clipboard.

Special characters can be entered using the Unicode escape **Ctrl-u**. The sequence starts by pressing **Ctrl-u**, then entering hex digits representing the character code, and is terminated with a space character or **Enter**. The Unicode coding can be obtained from tables provided on the internet, or from applications such as KCharSelect which is part of the KDE desktop. These are generally expressed as "`U + xxxx`" where the `xxxx` is a hex number. It is the hex number that should be entered following **Ctrl-u**. For example, the code for π (pi) is `03c0`. Note that special characters can also be selected and copied, or in some cases dragged and dropped, from another window.

Chapter 4

Using *Xic*

Xic has two basic operating modes: physical and electrical. In physical mode, one is editing the geometry of the mask patterns on the multiple layers used in the photomasks to manufacture the circuit. In electrical mode, one is editing an electrical schematic of the circuit or subcircuit represented by the cell. The schematic is used for documentation, and also for performing simulation of the circuit to verify performance. The schematic and layout can be interlinked to provide consistency verification. This is the purpose of the functions in the **Extract Menu**, to be described in Chapter 16.

A full design database typically consists of a hierarchy of cells. The top level or main cell usually depicts the entire chip. Subcells represent the bond pads, annotation, and major circuit blocks. The circuit blocks in turn have subcells representing more primitive circuit blocks, down to the gate level and below.

In *Xic*, one can edit any of these cells and their subcells at any depth in the hierarchy, as both physical layout and electrical schematic. The use of a hierarchical database is far more efficient and convenient than a flat database. The designer is encouraged to make liberal use of subcells rather than designing single, highly complex cells.

When a design is complete, i.e., when all electrical simulations and physical design rule checks have been performed, the physical part of the database can be submitted for processing. The exact mechanism varies with organization, but the physical-only (**Strip For Export** button in the **Export Control** panel from the **Convert Menu** active) GDSII, OASIS and CIF outputs provided by *Xic* are portable to any mask fabrication facility or foundry.

The user can switch between physical and electrical modes at any time, by pressing the **Electrical** or **Physical** button (whichever appears) in the **View Menu**. Sub-windows, brought up with the **Viewport** button in the **View Menu**, are individually switchable between schematic and physical views. The side menus differ somewhat between the two modes, and some menu commands operate a little differently.

The next two sections of this chapter provide an introduction to editing in physical and electrical modes. The remaining sections provide information on certain *Xic* operation modes and features, and are somewhat more advanced in nature. The following chapters provide detailed information on all of the menu command functions.

The new user should read the first two sections of this chapter, and practice using *Xic* while reading the help messages.

4.1 Physical Layout Editing

In physical mode, one arranges geometrical shapes on the various layers to produce a working circuit. One can also place subcells, which have been previously created. The knowledge of what shapes to place, and where, is dependent on the technology in use, and represents the essence of integrated circuit engineering. The user must be familiar with these fundamentals, as *Xic* is only a tool for application of this knowledge.

The basic primitive used by *Xic* is the box. Boxes are filled rectangular structures representing an area of opacity on the corresponding mask level. The **box** button in the side menu, with the rectangular icon, is used to create boxes. With the **box** button active, the user points to the two diagonal corners of the box desired in the drawing window, and a colored box will appear. The color and fill pattern are set for each layer in the technology file, and can be changed by the user with the **Set Color** and **Set Fill** buttons in the **Attributes Menu**. The layer can be selected by clicking on the desired layer in the layer table, which is arrayed near the bottom of the main *Xic* window. Note that when boxes created on the same level overlap, they are clipped or merged so as to not actually overlap. This increases the storage and retrieval efficiency of the database.

If the created box is too small or otherwise causes a design rule violation, a message will appear, if interactive rule checking is active. By default, all objects are checked for design rule violations when they are added to the database, though this can be set otherwise in the technology file or if the **Set Interactive** button in the **DRC Menu** is not active. Objects that "fail" are actually in the database, and it is the responsibility of the user to correct the error when it is flagged.

Boxes can be used exclusively to create a working circuit, however other structures are sometimes more convenient. Wires are fixed-width paths that are often used to make electrical connections. The **wire** button in the side menu allows the creation of wires, and the **style** button can be used to change or set the wire width and end style. The **wire** button has a sideways L-shaped icon. Every layer has a default wire width. To construct a wire, simply click on the points of the drawing window which correspond to wire vertices, and click the last vertex twice to end the wire. Note that the wire can zigzag at any angle, however the angles can be fixed to multiples of 45 degrees by setting the **Constrain angles to 45 degree multiples** check box in the **Editing Setup** panel from the **Edit Menu**. Also note that acute angles will most likely cause a design rule violation message to appear.

Polygons are constructed in a manner similar to wires, using the **polyg** button in the side menu. This button has a triangle icon. The polygon is constructed by clicking at each desired vertex location, and is terminated by clicking again on the first vertex. Polygons can have edges with arbitrary angles, which can be constrained to multiples of 45 degrees with the **Constrain angles to 45 degree multiples** check box in the **Editing Setup** panel. Again, acute angles are likely to cause design rule violations. Polygons are most useful for constructing rounded or off-angle shapes used in high frequency circuits. It is also slightly more efficient to use polygons rather than a collection of boxes.

With none of the geometry-creating buttons active, clicking on an object can cause it to be "selected". Only objects on layers that are selectable, as shown in the layer table, can be selected. A selected object will be outlined with a flashing highlight. Selected objects are used by many of the other commands. An object can be deselected by clicking on it a second time. The status window below the layer table will indicate the number of objects selected. Multiple objects can be selected at once by pressing and holding button 1, dragging the pointer, and releasing. A ghost-drawn rectangle will appear during this operation. Objects which overlap this rectangle will be selected (or deselected if already selected). All selected objects can be deselected with the **desel** button in the top button menu (above the main drawing window).

Once selected, an object can be deleted, either by pressing the **Delete** key, or by pressing the **Delete** button in the **Modify Menu**. The objects will disappear from the screen, and the database.

Almost any operation which modifies the database can be undone with the **Undo** button in the **Modify Menu**, which is equivalent to pressing the **Tab** key. The last 25 operation are saved, and can be undone. The **Redo** button, or equivalently **Shift-Tab** will redo the last undo. All of the undone operations are saved in the redo list, however the redo list is cleared after each new operation that is not an undo.

The **Stretch** button in the **Modify Menu** is used to modify the shapes or sizes of boxes, polygons, wires, and labels. By pointing at the edge or corner of a box, one can move that edge or corner to a new location. Similarly, polygon and wire vertices can be moved. Polygons and wires can also be modified with the vertex editor built into the **polyg** and **wire** command buttons. If a polygon or wire is selected before pressing the corresponding command button, the vertices of the selected object will be marked. The selected vertices can be deleted or moved, and new vertices added.

The **erase** button in the side menu has an icon consisting of a box with a corner missing. This button is used to delete parts of objects. One clicks twice, or presses and drags, to define a rectangle, which is ghost-drawn during the operation. This rectangular area will be cleared of fill from any box, polygon, or wire. Wires may not be entirely erased, as they are only cut at points where the central path crosses the erase box boundary.

The user may have already designed one or more cells using *Xic*, which are then available for use as subcells in the present layout. Subcells are called and placed with the **place** command button in the side menu. After pressing the **place** button, the **Cell Placement Control** pop-up will appear, which allows the user to select a cell to place from cells that have been placed previously, or to enter a new cell name to place. The cell name can be dragged from the **File Selection** panel or from the list pop-ups in the **File Menu**. In addition, the **List** pop-ups contain a **Place** button which will also set the name of the current "master" cell to be placed, and pop up the **Cell Placement Control** pop-up if it is not already visible. When the **Place** button in the **Cell Placement Control** pop-up is active, the current "master" will be "attached" to the mouse pointer, and instances will be placed at locations where the user clicks with mouse button 1 in the drawing. There is provision in the **Cell Placement Control** pop-up to define array parameters, so that an array of instances will be created rather than a single instance. The placement mode can be exited by pressing the **Esc** key, or by unsetting the **Place** button in the **Cell Placement Control** pop-up.

Once a physical layout is substantially complete, the layout is a candidate for batch design rule checking and extraction. These capabilities are described in detail in later chapters.

This brief introduction should convey the flavor of using *Xic* in physical mode. There are many more commands, and some of the commands introduced have additional features not mentioned. The best way to learn *Xic* is to use it, and read the on-line help available for the commands. After pressing the **Help** button in the **Help Menu**, pressing any command button will bring up a help screen describing the command. Reading the help and then trying the operation is the fastest way to learn. The help mode, and any command, can ge exited by pressing the **Esc** key.

4.2 Electrical Schematic Editing

The electical mode of *Xic* allows a schematic representation of the cell to be entered. This electrical representation is used to generate a SPICE file for simulation purposes, by *WRspice* or another simulator. The electrical representation can be generated or updated from the physical layout, if extraction has been properly set up, and can be compared with the physical representation to identify wiring errors.

The electrical representation of a hierarchy of cells follows the same hierarchy as the physical cells, for the most part. Physical cells that contain wire only, i.e., no devices or subcircuits, generally do not have an electrical-mode counterpart. Such cells are effectively flattened into their parents in the electrical representation. The physical implementation of devices can include structure from subcells. In this case, the electrical implementation of the device is in the electrical cell corresponding to the top-level physical cell containing the device geometry.

One does not need a physical representation in order to use electrical mode. In this case, *Xic* is used exclusively as a schematic capture front-end for *WRspice* or another SPICE-compatible simulator.

This section will focus on the mechanics of schematic entry and simulation using *WRspice*. The chapter on extraction (16) will provide detail on how the electrical and physical data can be made to interact.

To produce a schematic cell, one follows this basic outline:

1. Devices from the device menu or some other source are placed at various locations in the drawing. Also, subcircuits from the user's library are similarly added to the drawing.
2. The devices and subcircuits are wired together.
3. Properties are given to the devices, which designate component values, models referenced, or other information.
4. If the cell is to be used as a subcircuit in another schematic, connection points are added, and possibly a symbolic representation defined.
5. A SPICE file representing the present hierarchy can be generated at this point, or, if the circuit is top-level (not used as a subcircuit) interactive simulation using *WRspice* is possible.

The following sections will describe these steps in more detail.

A prerequisite for using electrical mode is basic knowledge of the SPICE syntax and SPICE file format. One does not need to be an expert, but a certain proficiency is assumed for such steps as property setting. It is recommended that users unfamiliar with SPICE skim the *WRspice* manual or other reference.

4.2.1 Placement of Devices and Subcircuits

Xic is distributed with a representative device library, which is contained in a file named `device.lib` found in the installation startup directory. This contains most if not all of the devices supported by *WRspice*, however it may be necessary to customize this file to the user's unique requirements. The format of this file is described in the appendix. The devices found in the device library file are those listed in the device menu, which is available while in electrical mode.

Devices can also be supplied in cell files, or from an OpenAccess database. For example, it is feasible to use devices from the `analogLib` library from a Virtuoso installation, or from a foundry design kit.

Xic usually starts in physical mode, though if given the `-E` option on the command line *Xic* will start in electrical mode. To switch from physical to electrical mode, press the **Electrical** button in the **View Menu**. *Xic* will reconfigure the side menu, and display the schematic for the current cell (if any). Pressing the **devs** button in the side menu will bring up a device menu which extends across the top of the main *Xic* window. There are two styles of device menu available. The default menu consists of an array of lettered buttons. Pressing button 1 while the pointer is over one of these buttons will cause a drop-down menu to appear, which consists of more buttons containing device names. The first letter of these devices is that on the original button. A device can be selected by releasing button 1 while the pointer is over the desired button.

A second device menu style consists of panels containing the names and schematic symbols of the various devices with perhaps a button with a right-pointing arrow, if the selections do not entirely fit on-screen. Clicking on the arrow button will show the devices which did not fit in the initial menu. This menu has the disadvantage of occupying a lot of screen space, but it may be easier for new users.

Both menu styles contain a button that switches to the other style of menu. The present style will be used until changed by the user. The style used is completely arbitrary, and simply a user-preference.

Clicking on one of the device panels in the pictorial menu, or releasing button 1 on a selection in the pull-down menu will attach the schematic symbol to the mouse pointer. Then clicking in the drawing window will leave instances of that device at those locations. Press **Esc** to exit this mode. This is the means by which devices are added to the circuit. New devices can also be produced by using a copy operation (a button 1 operation, or explicitly using the **Copy** command in the **Modify Menu**) from an existing device in the circuit.

The user may have already designed one or more circuits using *Xic*, which are then available for use as subcircuits in the present schematic. The details of how to create a "true" subcircuit will be presented shortly; for now, assume that such cells already exist. Subcircuits are called and placed with the **place** command in the side menu, in the same manner as subcells in physical mode. After pressing the **place** button, the **Cell Placement Control** pop-up will appear, which allows the user to select a cell to place from cells that have been placed previously, or to enter a new cell name to place. The cell name can be dragged from the **File Selection** panel or from the List pop-ups in the **File Menu**. In addition, the **List** pop-ups contain a **Place** button which will also set the name of the current "master" cell to be placed, and pop up the **Cell Placement Control** pop-up if it is not already visible. When the **Place** button in the **Cell Placement Control** pop-up is active, the current "master" will be "attached" to the mouse pointer, and instances will be placed at locations where the user clicks with mouse button 1 in the drawing. The placement mode can be exited by pressing the **Esc** key, or by unsetting the **Place** button in the **Cell Placement Control pop-up**.

Once devices and subcircuits have been placed in the drawing, they can be moved and copied as for physical cells. Not all of the transformations of physical mode are available, however, from the **xform** command in the side menu. Specifically, rotations are limited to multiples of 90 degrees, and there is no magnification capability.

4.2.2 Semiconductor Devices

The device menu contains symbols for the semiconductor devices supported by *WRspice*. These include diodes, bipolar and junction field-effect transistors, MESFETs, and MOSFETs.

Device	Description
dio	junction diode
npn	npn bipolar transistor
pnp	pnp bipolar transistor
njf	n-channel junction field-effect transistor
pjf	p-channel junction field-effect transistor
nmes	n-MESFET
pmes	p-MESFET
nmos	n-MOSFET (3-terminal)
pmos	p-MOSFET (3-terminal)
nmos1	n-MOSFET (4-terminal)
pmos1	p-MOSFET (4-terminal)

Unlike simple devices such as resistors and capacitors, which are fully specified by a value, these devices almost always require a model. The model is specified with a model property, which is applied to the device in the same way that a value property is applied to a simple device.

In order for *Xic* to include the model in the SPICE file, the model must be available to *Xic*. Device models are provided to *Xic* through a file read by *Xic* when the program starts. When *Xic* starts, it traverses the library search path, looking for model files. A model file is 1) a file usually named "`model.lib`", in which case the first such file is read, or 2) any file found in a subdirectory usually named "`models`" of a directory in the search path. The names assumed ("`model.lib`" and "`models`") can be changed in the technology file.

The files that contain the models consist of the `.model` lines for SPICE. These blocks are placed one after another, with no order assumed.

Perhaps the simplest way to add a model to *Xic* is through the `model.lib` file. A skeletal `model.lib` file is provided with *Xic*, in the startup directory. Models added to this file will be available to all users. If a copy of the `model.lib` file is placed in the current directory, (which is always searched first) then that file will be used instead. The first `model.lib` file found in the library search path will be used. This allows users to access their own custom `model.lib` file.

If large numbers of models are to be added, it may be more convenient to add a "`models`" subdirectory to one of the directories in the library search path. One may add a directory to the search path for this purpose. In the models subdirectory, add the files containing the SPICE models. The file names are unimportant, and all files found in the subdirectory will be searched.

Each model block starts with

`.model` *modname modtype ....*

The *modname* is an arbitrary word which designates the model, and this should be unique among all of the models *Xic* will find along the library search path. The *modtype* is the SPICE name for the model for a given device, as specified in the *WRspice* documentation. The remaining text consists of parameter value assignments as per the documentation. The *modname* should be used in a model property of the devices that are to use the model.

There are two different MOS device types: the nmos1/pmos1 devices contain stubs for all four nodes (gate, drain, source, and bulk). The nmos/pmos devices automatically connect the bulk node to global nodes named NSUB and PSUB, respectively. Most of the time, it is more convenient to use the nmos/pmos devices to avoid having to make explicit contact to the substrate nodes in the circuit, however one *must* remember to bias the NSUB and PSUB nodes. To do this:

If there is one or more nmos devices in the circuit:

1. Add a voltage source to the schematic.
2. Place a ground terminal on the negative terminal of the voltage source.
3. Place a tbar terminal device on the positive terminal of the voltage source.
4. Select the 'tbar' label of this terminal device.
5. Press the **label** button (side menu), and change the name from "tbar" to "NSUB".
6. Add a value property to the voltage source to set the substrate voltage. This procedure is described below.

If there is one or more pmos devices in the circuit:
Follow the same procedure above, however use "PSUB" as the name for the tbar device.

This will provide a dc bias voltage to the common connection of all of the nmos and pmos bulk nodes in the circuit. The value of NSUB is usually equal to the most negative supply voltage in the circuit, and the value of PSUB is usually equal to the most positive voltage in the circuit.

4.2.3 Wiring Devices and Subcircuits

Once the devices and subcircuits have been placed, wires can be added to make connections between them. This is not typically a two-step process, as most users build a schematic by mixing placement and wiring operations.

First, it should be stressed that connections do not always require wires, and in particular it is often most convenient to make connections between devices by abutment. Devices and subcircuits have specific local coordinates where a connection is possible. In a device, these are typically at the end of the wire stubs shown as part of the device symbol. In subcircuits, these are the terminal locations defined by the designer of the subcell, and can be made visible with the **terms** button in the side menu. When moving or placing a device, or creating a wire, visual feedback is provided when the mouse pointer is over a possible connection point. Connections can only occur at the connection points. The **Connection Dots** button in the **Attributes Menu** can be used to draw a dot at all connection locations.

The devices in the device menu should mostly be familiar to users of SPICE. There are special terminal "devices" that can be used instead of wires to provide interconnections. These are the "gnd", "tbar" and equivalent terminals. In the first case, the symbol is of a ground connection, and it provides exactly that. At least one point of every circuit must be grounded, or the SPICE simulation may fail. The tbar terminal is more general purpose. As it is, this terminal will tie all locations attached to such terminals together. This is a convenient way of distributing a power net, for example. If the name label of the tbar device is changed, then all locations attached to terminals with this name will form a *different* network. The easiest way to change the name is to click on the "tbar" label of an existing tbar device (thus selecting the label), then press the **label** button in the side menu. The user will be prompted for a new string. Once the new string has been entered, the label will be updated, and the terminal can be copied to other locations to from the network.

Remaining connections are made with the **wires** button in the side menu, which has an icon that looks like a sideways L. Before generating wires for connections, the user should make sure that the current layer is the "SCED" layer. Wires on this layer are electrically active. Wires created on other layers are for decoration purposes only, unless the WireActive flag is set for the layer.

Wires are used to connect the devices together by clicking on the vertex locations of the wires. The vertices must be on the contact points of devices and subcircuits, i.e., the ends of the connecting wire stubs of the devices, and the terminal locations of subcircuits. These vertices are created automatically in horizontal or vertical wire segments which cross over contact points.

One of the problems that some new users encounter is that contact is not made due to improper placement of wires in relation to device contact points. To reiterate the previous discussion, only the ends of the wire stubs of devices are "active", and these must physically coincide with a wire vertex. Although a vertex will generally be created if necessary in an intersecting wire, new users should form the habit of explicitly creating a vertex, by clicking on the contact point while creating the wire,

In electrical mode, the first layer in the layer table is a layer named "SCED". This is an active wiring layer, and by default only this layer can be used for electrically significant wires. The layer named "SPTX" is also active, in that labels on this layer are included in the SPICE text generated for the cell. Other layers are used for visual purposes only (such as color-coding the displayed property labels), or for temporary "storage" of parts of the circuit not in use. The **Chg Layer** button in the **Edit Menu** is used to change the layer of objects.

The additional layers can be used for anything, but serve the following purposes:

SCED	active wiring layer
SPTX	active label layer
NAME	device/subcircuit name property labels
MODL	device model property labels
VALU	device value property labels
PARM	device/subcircuit param property labels
NODE	terminal label
ETC1	general purpose
ETC2	general purpose

The **Connection Dots** button can be used to show dots at connection points. New users often appreciate the feedback provided by the **Connection Dots** button that a connection has been made. One has a choice of whether dots appear at every connection, or only at those likely to be ambiguous. When a wire is created, if it runs over a device terminal or a vertex of another wire while horizontal or vertical, a vertex is generated, which implies a connection. Two wires crossing do not connect, unless a vertex existed in one of the wires at the crossing point. Sometimes, is is desirable to remove a connection, or to enforce a connection of two crossing wires. This can be accomplished with the vertex editor available with the **wires** button. First, select the wire by clicking on it. After pressing the **wires** button, each vertex of the wire will be shown with a small box. Clicking on a vertex box will select that vertex, and allow the vertex to be dragged to a new location or deleted. In either case, the connection to an underlying vertex or device terminal will be broken. To add a vertex, click on the selected wire at the point where the vertex is to be added. A new vertex box will appear. If there is an underlying device terminal or wire vertex, a connection will have been established. If two wires cross with neither wire having a vertex at the crossing point, adding a vertex to one of the wires will automatically add a corresponding vertex to the second wire if the second wire is horizontal or vertical at the crossing point.

4.2.4 Adding Properties to Devices

Once the devices have been placed, device properties can be assigned. This is the method by which *Xic* knows the values, models, and other characteristics of the devices. Device properties are initially added with the **Property Editor** brought up by the **Properties** button in the **Edit Menu**. The **Property**

Editor contains a text window showing the properties of a selected device, if any. The features and capabilities of the **Property Editor** are rather complicated, and are described fully in the section of this manual (10.10) describing the **Properties** command in the **Edit Menu**. This section will describe some of the basic operations.

At this point there are four properties of interest: devref, value, model, and param. The purpose of the devref property is to hold the name of a device whose current is to be referenced. This is used by the current-controlled sources and switch devices only. The value and model are just different names for the same underlying text field, thus a device should not be assigned both a value and a model property. The param property will hold text for initial condition and parameter assignment.

The string for a device, which will be generated in SPICE output, has the generic form

device_name node_list [*dev_ref*] *model_or_value* [*parameters*]

The current-controlled dependent sources and switch require a devref property. This should not be used in other devices. Every device should have a model or value assigned. The parameter (param property) is optional, but may be needed for certain devices for particular types of simulation. It is also used to provide parameter values, such as the width or length of a MOSFET. This is where knowledge of the SPICE syntax is necessary, in order to know what parameters are required for a given device.

For simple devices such as resistors, only a value property is generally required. To apply a value property, with the **Property Editor** visible, click on the device to receive the property. The editor will list any existing properties, and the selected device will be highlighted. From the **Add** menu of the **Property Editor**, press the **Value** button, and enter the value on the prompt line, followed by **Enter**. A label showing the new value will appear next to the selected device.

The “value” can be just about any string, so it is important that this input have relevance to SPICE. The format of the numerical entries is as recognized by SPICE, in MKS units. One common error is to leave off the units, e.g., entering “`50`” for the value of a capacitor when the correct entry should be “`50fF`”. Of course, “`50e-15`” would suffice as well in this case.

The **Global** button on the **Property Editor** can be used to set the properties of several devices at once. The **Edit** button can be used to edit an existing property. Once a property has been assigned to a device, copies of the device will contain the same property, thus it may be preferable to assign properties in part early in the placement step, and generate copies of similar devices rather than placing new instances.

Once a property has been assigned, it can be changed with the label editor, thus the **Property Editor** needs to be invoked only for the initial assignment. To change the value of any editable property, select the label displaying that value (you can select properties in multiple devices). Then, press the **label** button in the side menu. This will prompt for a new value, and when given, all of the selected labels will be updated with the new value, and the underlying properties will have been changed.

4.2.5 Creating Subcircuits

In order for a cell to be a valid subcircuit, i.e., electrically active when placed into another cell, one or more contact terminal locations must be defined. This is accomplished with the **subct** button in the side menu. When this button is pressed, the user may click on contact points within the circuit to define contact locations. Only valid contact points can be selected, i.e., the point must fall on a wire vertex, or a contact point of a device or subcircuit. When a valid point is clicked on, a boxed digit will appear at the location, and a pop-up window will appear allowing the user to set the name and other properties of the terminal. If no name is given, *Xic* will use a default name.

Clicking on an existing terminal will start a move operation on the terminal, attaching its outline to the mouse pointer. Pressing the **Delete** key at this point will delete the terminal. Clicking on a terminal with the **Shift** key held, or double-clicking, will bring up the terminal editing window for the terminal, allowing modification of its properties.

The **terms** button in the side menu, when on, will display the terminal locations, as well as the terminal locations of subcells in the drawing.

Subcells will most often have terminals defined, which are the connections points to the cell. It is possible, though, that a subcell will have no terminals, if connection is made via global nets. Imagine a subcell containing only a capacitor, which is connected to global nets `vdd!` and ground. Adding an instance of the cell is equivalent to adding a decoupling capacitor.

It is possible, after an instance of a cell has been placed, to use the **Push** command to push into the new cell, and define additional subcircuit contacts, and pop back to the parent cell.

In some cases, it is preferable that the subcell be displayed as a symbol, rather than a schematic, when expanded. For example, if the subcell represents an AND gate, and there are many instances of the subcell, the drawing of the parent cell will appear much neater if the AND gate cell is represented by an AND symbol rather than its full schematic. One can define such a representation with the **symbl** button in the side menu.

On pressing the **symbl** button for a cell without a previous symbolic representation defined, the schematic will disappear, and the screen will be blank. One is free to use the objects from the **shapes** menu, wires, and labels, on any of the layers, to construct a symbol which will be displayed for that cell. When the new drawing is complete, the **subct** button should be pressed again. This will make the contact point indicators visible, however they will be in arbitrary locations. The user should move the terminals to where they belong in the symbolic representation, by dragging them with the left mouse button. Unlike in the normal schematic representation, the terminals can be placed anywhere. It is possible to copy terminals by holding **Shift** during the "move", so that the symbol may have multiple connection points for the same terminal.

New terminals can be added, or terminals deleted, only by returning to schematic mode, and similarly the schematic can be edited only by returning to schematic mode. The display status of the cell is set by the status of the **symbl** button when it was saved to disk, or last edited if it is still in memory.

4.2.6 Node and Device Naming

Xic will assign names and node numbers to the device, subcircuits and nodes in the circuit, by default. These will be unique numbers for each type of device and for each node. One problem, however, is that these numbers will change when the circuit topology is changed. Often, the SPICE output may be used by another application, that may need to access circuit node voltages, for example, in a predictable way. Thus, *Xic* has provision for assigning an immutable name to wire nets, and to devices and subcircuits.

By default, device names are assigned by *Xic* as the device key letter followed by an integer that *Xic* generates. This can be overridden by assigning a `name` property to the device. The procedure is identical to assigning the properties that we have discussed previously. The **Name** button in the **Add** menu of the **Property Editor** is used. Although the string that is entered as the name property can be anything, there are some very important constraints for correct SPICE output.

1. The first letter of the name must be the same (case insensitive) as the default name. This is the 'key' that identifies the type of device in SPICE.
2. The name should be a single word containing alpha-numeric characters only.

3. The name should be unique in the circuit.

Although *Xic* provides flexibility in assigning this property, SPICE simulations will fail unless these constraints are observed. Once the name property is assigned to a device, that name, rather than the default, will be used to reference the device. The name will appear in a label next to the device on-screen. As we have previously seen, the name can be modified subsequently with the label editor.

The procedure for assignment of names to subcircuits is identical. The ‘key’ letter for subcircuits is ‘`X`’.

The node mapping editor, which appears when the **nodmp** button in the side menu is pressed, is used to assign names to nodes. A “node” is SPICE terminology for a collection of one or more device and subcircuit terminals that are connected together. Each node is given a unique number by *Xic*, which is used as the node “name” in SPICE output. The node mapping editor allows the node to have an assigned name, which will be used instead.

Full information on the node mapping editor can be found in the section describing the **nodmp** command (7.11). Here, we will briefly outline its use. The left panel of the node mapping editor contains a list of the circuit nodes, with the left column containing the internal number, and the right column containing the assigned name, if any. Selecting an entry in this list will cause the device terminals for that node to be listed in the right panel, and these will be highlighted in the schematic. Pressing the **Rename** button will prompt the user for a name for that node. This can be any word consisting of alpha-numeric characters. This word will be used in SPICE output to designate the node, rather than the number.

4.2.7 Connectivity Overview

Thus far we have described the basic methodology for producing a schematic. Armed with this information, users can quickly produce schematics of simple circuits. However, a lot has been skipped over, including the use of multi-conductor nets and vectorized instances. This section will review the basic connectivity concepts, and introduce these new topics.

Devices and subcircuits generally have “pins” which are hot-spots in the drawing where connection can occur. These hot spots may or may not be marked in the device or subcircuit symbol or schematic. In any case, pressing the **terms** button in the electrical side menu will cause the display of terminal symbols at these locations.

The current cell will have its own terminal locations, if any have been defined with the **subct** command in the side menu. These will be the connections points to instances of the current cell.

Establishing connectivity in the schematic involves logically grouping the device, subcircuit, and cell terminals that should be connected. Each such group is termed a “net”. There are a number of ways to define this grouping.

1. Most commonly, a wire is placed by the user using the **wire** command in the side menu. To establish connectivity, a vertex of the wire must be at a connector hot-spot. If the **dots** display is enabled, a dot may be shown at the connection points.

2. Connection points whose hot-spots are placed at the same location will be connected.

These two methods illustrate connection by location. It is also possible to use connection by name. For this, one must use named nets. Looking ahead just a bit, it is possible for a net to be scalar (single conductor) or multi-conductor. The type of net is described by the name, which is interpreted as a "net expression", which is a syntax which allows detailed definition of the conductors in the net.

There are several ways by which a net can acquire a name.

1. Nets connected to named cell terminals will have the same name as the cell terminal, but only if the terminal has an applied name. Names can be given to cell terminals with the **subct** command in the side menu.
2. A scalar (single conductor) net can be assigned a name with the **Node (Net) Name Mapping** panel brought up with the **nodmp** button in the side menu. This name has priority over the "candidate names" applied with wire labels or terminal devices.
3. A candidate net name will be supplied by associated labels of wires in the net. A label is given to a wire through the following procedure.
   - In electrical mode, select a single wire, which shall receive a name.
   - Press the **label** button in the side menu.
   - Type the label text in the prompt line, and press the **Enter** key.
   - The label is ghost-drawn and attached to the mouse pointer. Resize or rotate the label if desired, and click in the drawing near the selected wire to place the label. This completes the operation.
4. A candidate net name can also be supplied by placing a terminal device from the device library in contact with the net. The device library provides several terminal styles. Each has a label that can be edited to apply a net name. Once placed, the label can be selected, the **label** button pressed, and new label text entered.

A scalar net may have multiple "candidate names", and each can be used to establish connections by name. However, the single name chosen to represent the net in netlist output will be the name that comes first in alphabetical order.

Nets that otherwise appear disjoint but have a common name are actually connected. This illustrates connect by name. In fact, it is possible to draw perfectly good schematics without using wires, by using terminal devices only. The schematics produced by *Xic* from SPICE files or physical extraction use this approach.

Xic supports multi-conductor wire nets in schematics, using a syntax and methodology that should be familiar to users of Cadence Virtuoso. The net name uses a syntax which describes the net. Unnamed nets will assume the characteristics of connected terminals.

There are three types of net.

Scalar nets

Single-conductor "scalar" nets provide the basic connectivity description in a schematic, and are the only electrical nets that may have a counterpart in the physical layout.

A scalar net name consists of a simple name, or an indexed vector name, in a format to be described.

Vector nets

A vector net contains multiple conductors, accessible as indices in a range, with a common base name. A name specifying a vector net may have the form

basename[*start*:*end*]

The *start* and *end* are non-negative integers. The two colon-separated numbers provides a range of subscripts which identify the individual conductors, or "bits", of the net.

For example, the vector net "`foo[3:0]`" consists of four conductors, in order "`foo[3]`", "`foo[2]`", "`foo[1]`", and "`foo[0]`". Note that the range values can be ascending or descending.

In *Xic*, the square brackets can be replaced by `<...>` or `{...}`. That is, for subscripting in *Xic*, square brackets, curly brackets, and angle brackets are equivalent. This documentation will use square brackets.

Vector nets differ fundamentally from scalar nets in *Xic* in that they simply reference scalar nets. The scalar nets actually provide the electrical connections, and the correspondence between layout and schematic. The vector and multi conductor nets in general simply provide an organizational framework for the scalar nets.

In particular, this requires that each "bit" of a vector net have an existing scalar net of the same name. In the example above, for the vector net `foo[3:0]` to be valid, the individual scalar nets `foo[3]` etc. must exist.

Bundle nets

A bundle net is a net of nets. Its name is a net expression consisting of comma-separated names of scalar and vector nets. Some examples would be

```
a,data[0:7],addr[2]
b0,b1,b2
```

These are simple cases of a net expression which describes the conductor sequence of a general net. Net expressions and vector expressions may be familiar from Cadence Virtuoso, and in fact the same operations and syntax are supported.

4.2.8 Net and Vector Expressions

The name of a net is parsed as an expression using a set of rules to be described. The result of this interpretation is that each conductor ("bit") of the net has a well-defined name, which is associated by name with all other nets in the cell with bits of a matching name.

We say "matching" rather than "the same" as *Xic* will ignore the different subscripting characters. In *Xic*, square, curly, and angle brackets are accepted for subscripting, thus the forms `foo<2>`, `foo[2]`, and `foo{2}` are equivalent ane can be freely intermixed in the design.

A net expression consists of one or more comma-separated *terms*.

net expression = *term*[`,`*term*]...

A *term* has the general form

subterm = *name*[*vector expression*]
multiplier = `[*`*N*`]`, or
multiplier = *N*`*`
term = [*multiplier*]*subterm*, or
term = [*multiplier*]`(`*term*[`,`*term*]...`)`

The basic element of a *term* is a *subterm*, which consists of a name optionally followed by a *vector expression*. The *name* is an alphanumeric text name. The *vector expression* represents subscripting to be described.

An optional *multiplier* can prefix the *term*. This is an integer N, and a literal asterisk, in one of the forms shown. Here, the literal square brackets can be replaced by curly brackets or angle brackets equivalently. Both forms of the multiplier prefix are equivalent. The effect of the multiplier is to repeat what follows N times.

The second form of the *term* allows for a list of *terms*, separated by commas and enclosed in parentheses. The commas and parentheses are literal. This allows the multiplier to cause repetition of the group of terms.

The multiplier provides a shorthand way to express repetitions, but is not required. Below are some examples and equivalences.

```
3*A = A,A,A
2*(A,B) = A,B,A,B
2*(A,2*B) = A,B,B,A,B,B
```

In each case, the shorthand on the left is equivalent to the ordering on the right. The `A` and `B` are scalar conductor names. The third line above, for example, describes a six-conductor net with the net bits connected to either net `A` or `B` in the order shown.

A *vector expression* represents a sequence on integers, each representing a conductor index.

bit = N
range = $N : M[: S]$
postmult = `*`N
vector expression = `[`*bit*|*range*[*postmult*][`,`...]`]`
vector expression = `[(`*vector expression*[`,`...]`)`[*postmult*][`,`...]`]`

Again, where literal square brackets are shown, curly brackets and angle brackets are equivalent in *Xic*. The elemental decomposition of a vector expression is a comma-separated list of non-negative integers. A *bit* constitutes one such integer. A *range* is specified by two or three colon-separated non-negative integers. In the simplest and most common form, the range consists of two integers and represents the two integers and all intermediate integers, in order. If a third integer is given, this represents the increment. The number sequence consists of the start value, and multiples of the increment, terminating at the final value that would not fall outside of the range. Note that the increment is always a positive value, whether the range values are decreasing or increasing. Below are some examples.

```
[3:0] = [3,2,1,0]
[3:0:2] = [3,1]
[1:4] = [1,2,3,4]
[1:4:4] = [1]
```

Either can be followed by a *postmult* multiplier, which causes each element of the sequence to repeat.

```
[0*2] = [0,0]
[3:0*2] = [3,3,2,2,1,1,0,0]
[1:4:4*2] = [1,1]
```

The final form illustrates use of literal parentheses and commas to associate a list of vector expressions to a post-multiplier. The entire list will be repeated. The parentheses can be nested to arbitrary depth.

```
[(1,3:5)*3] = [1,3,4,5,1,3,4,5,1,3,4,5]
[(1,(2,3*2)*2,4:6)*2] = [1,2,3,3,2,3,3,4,5,6,1,2,3,3,2,3,3,4,5,6]
```

4.2.9 Vectored Instances

Device and subcell instances can be scalar or vectorized. By giving an instance a range property with the **Property Editor** from the **Edit Menu**, the instance will become vectored. The single schematic representation in the drawing of a vectored instance actually corresponds to multiple "bit" instances. This can greatly clarify schematics with repeated circuit blocks.

The connections to a vectored instance are all multi-conductor nets (assuming that the array range contains more than one element).

4.2.10 Connection Rules

The following rules are applied when connecting by location.

1. Any named scalar net can connect to any other named (or unnamed) scalar net. A scalar net can have any number of associated names, each of which is a valid target for connect by name.

2. If a scalar net connects to a non-scalar net, the scalar bit will connect to each bit of the non-scalar net.

3. A net connecting to a vectored instance terminal must have a width equal to one of the following:

   - The total connection width, given by the pin width multiplied by the vector instance width. For example, suppose that the instance is arrayed `[0:3]` and the pin is `A[0:1]`. Suppose that the connecting net is `net[7:0]`. Then, all is well as the widths match, and connections will be as shown.

     `net[7]` = `X[0]A[0]`
     `net[6]` = `X[0]A[1]`
     `net[5]` = `X[1]A[0]`
     `net[4]` = `X[1]A[1]`
     `net[3]` = `X[2]A[0]`
     `net[2]` = `X[2]A[1]`
     `net[1]` = `X[3]A[0]`
     `net[0]` = `X[3]A[1]`

     If the widths do not match, a warning will be issued. *Xic* will connect what it can, in an order like that above, but some bits will remain unconnected.

   - The pin width. In this case, a virtual multiplier prefix is applied to the net. For the example above, but with `net[1:0]` that matches the width of `A[0:1]`, the connections would be

     `net[1]` = `X[0]A[0]`, `X[1]A[0]`, `X[2]A[0]`, `X[3]A[0]`
     `net[0]` = `X[0]A[1]`, `X[1]A[1]`, `X[2]A[1]`, `X[3]A[1]`

   - The width is one (scalar net). In this case, all of the instance pin bits would connect to the same scalar net.

4. Named multi-contact nets cannot connect to incompatible nets. Two named nets are "compatible" if one is a "tap" of the other. This will be described in the next section. Violations generate an error message and no connection is made.

4.2.11 Tap Wires

The concept of tap wires may be familiar from Cadence Virtuoso. Tap wires are fully supported in *Xic*.

A wire is considered to be a "tap" of another wire if every bit in the first wire is included in the second. Note that they may have very different bit order.

If a wire is a tap for another, then the two wires are allowed to connect. Note, however, that the visual connection may serve no real purpose, as the bits are already connected by name. However, the visible indication of connectivity may make the schematic more readable. The tap wire will allow connection to a subset of the conductors in the wire being tapped.

An interesting special case is when the wire being tapped is a pure vector. In this case (only), the tap wire label need not include a name, but only a vector expression. Also in this case, a connection is required. Then, the tap wire will obtain the name from the wire being tapped.

For example, suppose that we have a net `data[0:3]`, and we want to connect `data[0]` to a scalar instance pin `A`. If we connect the `A` pin directly to the `data[0:3]` wire, all four bits of the wire would be connected to `A`, which is not what we want. Instead, create a new wire, connected to the original wire and to `A`. Give the new wire a label "`[0]`". This becomes a tap wire, connecting `data[0]` to `A`.

4.2.12 Generating Output and Running Simulations

Once the device properties have been entered, the user can export the circuit for further analysis. The **deck** command in the side menu can be used to produce a SPICE file of the current hierarchy. If the *WRspice* program is accessible, the **run** command in the side menu can be used to initiate analysis. The user will be prompted for a SPICE analysis string, and the simulation will run. A small window will appear that will inform the user when the analysis is complete.

After *WRspice* analysis, circuit variables may be plotted. The **plot** command in the side menu allows the user to click on circuit nodes to plot. After each click, the corresponding node is added to the string shown on the prompt line. This string can be edited manually in the usual way, if necessary. Pressing **Enter** will terminate the string, and the plot will be displayed on-screen. The **iplot** button works similarly to the **plot** button, though the plot will be generated dynamically during simulation on subsequent runs. Plotting is available only through the *WRspice* program.

Once properties have been entered, they are easy to alter without the use of the **Properties** command. The **label** button in the side menu is used primarily to add annotation to the drawing. However, if a label is selected before pressing the **label** button, the existing label can be edited, rather than a new label created. If the selected label is one of those created for a property, then that property can be altered merely by editing the label. Thus, to change a property of a device, click on the label to select it. Then, after pressing the **label** button, enter the new text. The circuit can then be re-simulated with the altered parameters.

One feature of *Xic* is the use of hypertext. This is most evident when using the **plot** command. When the user clicks on a circuit node, the name of that node is entered, in color, on the prompt line. Note that when using the arrow keys to move the prompt text cursor across a node name, the cursor widens to underline the name, and the name otherwise behaves as a single character. The name shown is a link to the internal database, and has the property that if the node number assigned to that contact point changes (it may, if the circuit is modified, as it is by default randomly assigned) the string will automatically be updated to the new node number.

When creating a label, clicking on a connection point in the drawing, for example, will enter a hypertext link to the node into the label. The label will always display the correct node number or name for the node. This is the means by which node labels should be added to the drawing.

The same feature can be used to set up specialized spice output. Suppose one wishes to use the **save** command in SPICE. A "spicetext" label can be created, where the nodes to be included in the save are inserted in the label by clicking on the drawing. When a SPICE file is produced, the contents of the "spicetext" labels is added to the deck. The resulting save command will always save the clicked-on nodes, whether of not the actual internally generated number changes.

The "spicetext" label is simply a label where the first word is "spicetext" or "spicetext$N$" where N is an integer. These labels have the property that any text following the "spicetext" keyword is added to the SPICE output verbatim. The optional integer that follows "spicetext" determines the order of appearance of the lines, where no integer is equivalent to 0. This is the mechanism for placing arbitrary text into the SPICE output.

This has been a brief introduction to the use of *Xic* in electrical mode. There are numerous commands and features, and many of the commands mentioned have not been fully described. The easiest way to learn *Xic* is to use it. After switching to electrical mode, press the **Help** button in the **Help Menu**. Pressing any button will bring up a description of that command. Press **Esc** to exit help mode.

If a cell has both a physical layout and electrical schematic, there is provision for verifying consistency of the two representations by performing layout vs. schematic (LVS) testing. This is one of the functions which can be found in the **Extract Menu**, and the process is described in Chapter 16.

4.3 Cell Organization and Libraries

Xic provides several methods by which collections of cells can be organized.

- *Xic* makes use of a search path for file names given to *Xic* which do not have a directory path prepended. A search path is a list of directories where *Xic* searches for a named file. If the file name contains a full path, that path will be used to obtain the file. If a file name has a relative path, *Xic* will look for the file relative to each of the directories in the search path. The search path can be set in the technology file, or by setting the Path variable with the !set command. The current path can be examined by entering "`!set`", which pops up a list of the currently defined variables, including Path. The directories are searched in left-to-right order.

- *Xic* accepts library files. These are text-mode files which contain references to cells and other libraries, and may contain cell definitions. If a library file is "open", cell names referenced or defined in the library will be resolved through the library, before resolving through the search path. The name of a cell reference in a library is the name by which the cell will be added to *Xic* memory, which can be different from the name by which the cell is stored on disk. The fact that a library can reference other libraries allows a hierarchy to be established for accessing cells, independent of the search path.

  The **Libraries List** button in the **File Menu** brings up a panel which lists the currently open libraries, and provides command buttons for performing basic manipulations on libraries, including opening/closing, viewing content, and opening cells.

- Cells contained in archive files can be randomly accessed from the file, thus these files can be used for archival purposes. The **Contents** button in the panel brought up by the **Files List** button in the **Files Menu** will display the cells contained in these files. The **Contents** button will also list the contents of library files. Individual cells (and their subcells) can be opened for editing or placement through this panel. Also, when giving a name to the **Open** command, or the **place** command in the side menu, one can give two names: the name of an archive file and a space-separated name of a cell in the archive. That cell will be opened. If the cell name is not given, the top-level cell in the archive is opened.

The strategy used to organize cells is highly dependent upon the user's needs and preferences. Below are some recommendations which are probably suitable for most applications.

- Keep the search path short. This can usually consist of two directories: the current directory (".") listed first, and a root directory for the user's design files. The search path is most conveniently defined in the technology file, with the Path keyword. The search path has the disadvantage that all components are visible at all times. If a cell name appears more than once in the search path, only the first instance will be found, unless the full path is given. Libraries, on the other hand, can be opened and closed easily, changing the accessibility of the contents.
- Use hierarchies of libraries rooted in the search path to organize cells. One can open only the libraries in use, preventing loading of cells unexpectedly.
- Place collections of cells to be referenced through libraries in separate directories not in the search path. Alternatively, the *Xic* cell definitions can be incorporated directly into the library file. The cells can otherwise be kept as individual cells of any compatible format, or combined into a single archive file.

Library files have a simple format which allows the user to easily create and customize them with a text editor. There is a !mklib command in *Xic* which can create a new library or append to an existing library references to cells in the current editing hierarchy or cells in a given archive file.

If one clicks on a reference in a library content listing which points to another library, without a resolving "*cellname*", a second content window appears providing a listing of the second library's references. Thus, when constructing library files, one should use an easily recognizable name for browsable references to other libraries. This is natural, if the file name is used as the reference name, and the filename has a "`.lib`" extension as is recommended.

4.4 Batch Mode

Xic has a batch mode of operation, where *Xic* will start without graphics, run commands, and exit. Batch mode is signaled by giving the -B option in the command line, in one of the following forms:

-B*scriptfile*[,*param1*=*value1*][,*param2*=*value2*]...
-B-*command*[@*arguments*]

In the first form, the path to a file containing *Xic* script statements immediately follows "-B" with no space. The statements in the script file will be executed after the first input file is loaded. If no input file is given on the command line, the script will be executed after the default "noname" cell is loaded.

It is possible to pass parameters to the batch-mode scripts from the command line. The comma is used as a delimiter. Commas in the line that remain in single or double quotes *after* the shell has treated the line are not taken as separators. The entire construct should not have any embedded white space, except when single or double quoted as part of the *values*.

The *param1*, *param2*, etc. are the names of variables that will be defined in the execution context of the script. These variables will be set to *value1*, *value2*, etc. The values are numbers, strings, or executable text. Values that contain white space must be quoted, but note that the shell will strip the quote marks, so that a string constant should be single and double quoted as shown below.

Example

```
xic -Bmyscript,p1=1.234,p2='"a string"',p3="p1 + 1"
```

This translates into the virtual addition of three lines to the beginning of the script:

```
p1 = 1.234
p2 = "a string"
p3 = p1 + 1
```

In the second form, the "`-B`" is immediately followed by another '`-`' and one of the command keywords listed below. After the first cell is loaded (or "noname" if no input file was named in the command line) the command will be executed. The recognized commands are listed below.

The command name can be immediately followed by an argument string that begins with the '`@`' character. The arguments are specific to the command. Multiple arguments can be separated by '`@`' characters, or by white space if quoted.

The `.xicstart` file is read and executed (if it exists) before the first cell is loaded, and all other initialization is performed in the normal sequence. The commands below are simple shortcuts to common operations. If unavailable options are required, then these can either be set in a `.xicinit` or `.xicstart` file, or the first form of the `-B` option should be used.

`tocgx`, `tocif`, `togds`, `tooas`, `toxic`

These write the hierarchy under the current cell to CGX, CIF, GDSII, OASIS, and native cell files, respectively. They perform file conversion by reading a file into *Xic*, then writing it out in the specified format. The *FileTool* utility and `-F` command line option provide a far more powerful format translation capability.

The default name for the file written is the name of the current cell, suffixed with "`.cgx`", "`.cif`", "`.gds`", and "`.oas`" for the four archive file formats. Native cell files always have the same name as the cell contained.

These commands can take the following options. The options are separated from the command name and from one another by '`@`' characters, and consist of a single character identifier, an optional '=' character, and a value.

`o=`*outfile*

The *outfile* is the name of the file to be generated. If not provided, the file name will be the name of the top-level cell suffixed with an extension appropriate for the format. In the case of `toxic`, the *outfile* is a path to a directory where the cell files will be created.

`s=`*scale*

The *scale* is a floating point value from 0.001 to 1000.0 which applied when the file is written.

`l=+|-`*lname*[`,`*lname* ...]

This option specifies a list of layer names. The first character in the list is a `+` or `-` to indicate that only the listed layers will be output, or that all layers except the listed layers will be output, respectively. Immediately following is a layer name, optionally followed by additional layer names separated by commas.

`e`[*N*]
The letter '`e`' can be immediately followed by an integer 0–3. This sets the empty cell filtering level, as described for the **Format Conversion** panel in 14.10. The values are

`e` or `e1`	Use both pre- and post-filtering.
`e2`	Use pre-filtering only.
`e3`	Use post-filtering only.
`e0`	No empty cell filtering (no operation).

`f`
This flag option indicates that the output will contain a flat representation of the cell hierarchy. If the `w` option is given, only objects that overlap the window area will be present in output. This option will not work with `toxic`.

`w=`*l*,*b*,*r*,*t*
This specifies a rectangular area, in microns, for use when flattening.

`c`
This flag indicates that when flattening with a window (both `f` and `w` options also given) objects will be clipped to the window boundary in output.

Example:

```
xic -B-togds@o=file1.gds@w=100,200,200,300@fc@l=+0600 myfile.gds
```

This will create `file1.gds`, containing objects on layer 0600 within the window area, flattened and clipped. Note that the `@` separation character is actually optional after flags, and other options which are not lists or strings.

`drc`
Design rule checking is performed, and results are written to a log file.

There are optional arguments that can be provided, separated from the command name and from each other with '`@`' characters.

`w=`*l*,*b*,*r*,*t*
This provides an area, given in microns, of the top-level cell where checking will be performed. The value consists of four comma-separated floating-point numbers. If not given, the entire cell will be checked.

`m=`*maxerrs*
This provides the maximum batch-mode error count, checking will terminate when this count is reached. The *maxerrs* is an integer 0–100000, with 0 indicating no limit. This will override the maximum error count set in the technology file, if any.

`r=`*level*
This sets the error recording level to use when checking. The *level* is an integer 0–2. These correspond to recording one error per object, one error of each type per object, or all errors. This will override the recording level set in the technology file.

`d`
This a flag, not followed by an '=' sign or value. If given, the file which was the source of the current cell will be deleted from the disk when DRC completes. This facilitates cleaning up temporary files, but obviously should be used with care.

In batch mode, the log files for reading and writing of files are written to the current directory.

4.5 Server Mode

Xic has the capability of operating as a daemon process, servicing requests for processing of design data. This allows *Xic* to be used as a back-end for automation systems designed by the user or third parties.

To start *Xic* in server mode, the `-S` option is used, as

`xic -S`[*port*]

This causes *Xic* to start without graphics, go to the background, and listen to a system port for requests. The port number used can be provided on the command line immediately following the "`-S`". If not given on the command line, the "`xic/tcp`" service is queried from the local host. This will come up empty unless the "`xic/tcp`" service has been added to the host database, usually by adding a line like the following to the `/etc/services` file:

```
xic           6115/tcp   #Whiteley Research Inc.
```

where the port number 6115 is replaced by the desired port number. If there is no port assigned for "`xic/tcp`", port 6115 is used, as this is the IANA registered port number for this service.

If the XTNETDEBUG environment variable is defined when *Xic* is started in server mode, a debugging mode is active. *Xic* will remain in the foreground, but will service requests while printing status messages to the standard output. This may be useful for debugging. If the `dumpmsg` command is given, *Xic* will print the text of messages received on the terminal screen, enclosed in '—' symbols to delineate the text. The command `nodumpmsg` can be given to turn off the message printing. This can be a useful feature for debugging a client-side program which is communicating with *Xic*.

The user's application should open a socket to this port for communications. Up to five channels can be open simultaneously.

All transmission to the server is in ASCII string format, however replies are in a binary format, and are likely to be invisible or gibberish in a text-mode connection such as `telnet`. However, the `telnet` program can be used to connect to the *Xic* daemon, and can be used to give simple commands, such as the `kill` command. After starting the daemon, one types

`telnet` *hostname port*

where *hostname* is the name of the machine running the daemon (one can use "`localhost`" if running on the local machine). The *port* is the port number in use by the daemon.

An example file `xclient.cc` is available which provides a demonstration of how to interact with the *Xic* daemon through a C/C++ program. This file can be found in the examples directory of the *Xic* installation.

Communication can also be established through use of the example `xclient.scr` script, which illustrates use of script functions to implement a client within *Xic*.

While the server is working on a task, the server is sensitive to interrupts. An interrupt will cause the server to abort the current task and begin listening for new instructions. The interrupt handling works about the same as in graphical mode when the user types **Ctrl-c**, though there is no confirmation prompt — the task is always aborted. There may be a short delay before the interrupt is recognized.

Interrupts can be sent to the server by sending an interrupt ("`INT`") to the process number of the server with the Unix `kill` command. The server socket will also raise an interrupt if out of band (OOB) data are received. Thus, the client can send a single arbitrary byte of OOB data to generate an interrupt. The Unix manual pages describe the concept of OOB data.

The text expected by the daemon is in the form of statements which can be understood by the script interpreter, i.e., script lines. In addition, there are a number of special control commands.

As more than one connection can exist at the same time, commands from one connection can dramatically alter the environment seen by the other connections, including clearing of data and killing the server. Though the connections are separate, they should be considered as multiple windows into a single processing environment rather than separate processing environments.

Generally, when the last connection closes, all data within the server will be cleared and its state reinitialized, though this can be suppressed, allowing persistence of state and data.

The server may be used as a "geometry server", providing compressed representations of the geometry in cells, by layer, as from a Cell Geometry Digest (CGD). A connection object can be linked to a Cell Hierarchy Digest (CHD), allowing operations with the CHD to obtain geometry through the server. This would reduce memory use on the local machine, assuming that the geometry is stored on a remote server.

The built-in non-script commands are described below. All other input should be parsable by the script parser, except that lines that start with '#' are not allowed, so no comments or preprocessor directives are allowed.

All transmissions to the server are readable ASCII text, using standard network "`\r\n`" line termination. Replies from the server are in a binary form described below.

After each line of input is given, the server returns a message giving the data type and possibly the data for each script command. Most script functions return some value. Assignments return the value assigned. A variable name returns the value of that variable, if the variable has a known type. The default mode is to return only the data type code, which minimizes the network overhead. Optionally, the `longform` command can be applied, in which case the data are returned. Note that this can be arbitrarily large for some data types.

`close`
: This will close the connection to the daemon, and is the normal way to end a session. If no other connections are open, the daemon will generally clear the database of all cells and otherwise initialize itself to a clean state for the next connection (effectively calling `reset` and `clear`, see below), though this can be suppressed with `keepall` (see below). The daemon will continue listening for new connections.

`kill`
: This will close the connection and cause the server to exit.

`reset`
: This command will reset the script parser to its initial state, exiting from any control block in effect and deleting any script variables that may have been defined previously. This will affect all open connections.

`clear`
: This will clear the server database of all cells, and delete any layers that were not initially read from the technology file. This is equivalent to calling the `ClearAll` script function. This will affect all open connections.

`longform`
: After each line of script input is given and the line processed, a response message will be returned

based on the computed result from the line, if any. The user has a choice of receiving a very brief reply, giving only the response code - an integer which indicates pass/fail and the type of computed data, if any. The other choice is to actually return the data along with the response code. The data can be arbitrarily large.

The default return is "shortform" which does not transmit the data values. Giving this command switches to the mode where values are returned, for the present connection only.

`shortform`
When given, subsequent replies fro the present connection will use the short form for returned data, which consists of only the data type code. This is the default.

`dumpmsg`
When given, the text of subsequently received messages from the present connection is printed, surrounded by vertical bar ('`|`') symbols, on the standard output, meaning that the text will appear in the `daemon_out.log` file in normal operation. If the server is running in debugging mode (the XTNETDEBUG environment variable was found when the server started), this text will be printed on the console window.

`nodumpmsg`
This turns off the printing of received messages if `dumpmsg` was given. It has no effect otherwise, and applies only to the current connection.

`dieonerror`
Ordinarily, if the client crashes or there is a connection failure, the server will simply reset itself and continue waiting for new connections and handling other existing connections. If `dieonerror` was given, the server will instead exit on failure of the current connection.

`nodieonerror`
This will undo the effect of `dieonerror`, if `dieonerror` was given, and has no effect otherwise. It applies only to the current connection.

`keepall`
Ordinarily, when the server receives a `close` command, and there are no other connections open, the interpreter context is reset, the cell database is cleared, and other steps are taken to provide a clean environment for the next connection. If this command is given, all of this will be skipped, so that the same context and environment will be available to the next connection. This is a single flag which can be set or reset from any connection, but applies to all connections.

`nokeepall`
This will undo the effect of `keepall`, if `keepall` was given, and has no effect otherwise. This can be given from any connection, and applies to all connections.

`geom` [*chd_name*] [*cellname*]
The `geom` command implements the "geometry server", and unlike the other built-in commands this is an actual function and does not affect the interface state.

Information from Cell Geometry Digests saved in server memory is made available through this interface. The `OpenCellGeomDigest` script function can be used to create CGDs in the server, and of course the target layout file must be accessible to the server.

All of the arguments that follow "`geom`" are optional, though arguments to the left of a given argument are required. Below are the accepted forms and returns. In all cases, the actual data are returned, as with `longform`.

`geom`
: If no arguments are given, the reply is a space-separated string listing of CGD access names found in the server. If an access name contains white space, it will be quoted.

`geom ?` *cgd_name*
: This form will return the string "`y`" if *cgd_name* is the access name of a CGD in memory, "`n`" if not found.

`geom` *cgd_name*
: The argument is taken as an access name of a CGD in server memory. The return is a string containing space-separated cell names found in the indicated CGD.

`geom` *cgd_name* `-?`
`geom` *cgd_name* `?-`
`geom` *cgd_name* `-`
: The argument is taken as an access name of a CGD in server memory. The return is a string containing space-separated cell names that have been removed from the CGD.

`geom` *cgd_name* `?` *cellname*
: This form will return the string "`y`" if *cgd_name* is the access name of a CGD in memory, and *cellname* is found in that CGD. The string "`n`" is returned if the CHD access name matches a CGD name, but the *cellname* is not found in that CGD. An empty string is returned otherwise.

`geom` *cgd_name* `-` *cellname*
: if the *cgd_name* and *cellname* match a CGD and cell, that cell will be removed from the CGD, and resources freed. However, the cell name and its status as having been removed is retained. This will return the string "`y`" if *cgd_name* is the access name of a CGD in memory, and *cellname* is found in that CGD (and removed). The string "`n`" is returned if the CHD access name matches a CGD name, but the *cellname* is not found in that CGD. An empty string is returned otherwise.

`geom` *cgd_name* `-?` *cellname*
`geom` *cgd_name* `?-` *cellname*
: These forms will return the string "`y`" if *cgd_name* is the access name of a CGD in memory, and *cellname* has been removed from that CGD. The string "`n`" is returned if the CHD access name matches a CGD name, but the *cellname* is not in the removed list for CGD. An empty string is returned otherwise.

`geom` *cgd_name cellname*
: If two arguments, they are taken as the CGD access name and a cell name in the indicated CGD. The return is a string consisting of space-separated layer names of layers in the cell that contain geometry.

`geom` *cgd_name cellname* `? layername`
: This form will return the string "`y`" if *cgd_name* is the access name of a CGD in memory, and *cellname* is found in that CGD, and *layername* the name of a layer found in that cell. The string "`n`" is returned if the CHD access name matches, but either *cellname* or *layername* is not found. An empty string is returned otherwise.

`geom` *cgd_name cellname* `layername`
: With this form, the return value is the compressed string representing the geometry. These data have a unique return class, described in the format documentation below.

The normal way to terminate a session with the server is to issue the `close` command. Unless `keepall` is in effect, if there are no other open connections the server will be cleared and reinitialized. The clearing and reinitialization is equivalent to giving the `reset` and `clear` commands, which can be given at any time from any connection, and affects all connections. If the `keepall` command was in effect, the server will not be reset and cleared before the connection is closed, thus its state will be retained for the next connection. If there is a communications error, the server will exit if `dieonerror` was in effect for the affected connection, otherwise the behavior will be the same as for a `close` operation.

There is quite a bit of internal server state that is not reset to any preset value between connections. Examples are the mode (physical or electrical) and the status of variables set with the **!set** command or `Set` function. Thus, when writing scripts for execution by the server, it is important to explicitly initialize any such state or variable.

The `ReadReply` and `ConvertReply` script functions can be used the to handle server responses when the client is implemented as a script. For other applications, the user will have to write a parser, perhaps using the code from the `xclient.cc` example. Whiteley Research can provide assistance to users who need to develop this capability.

4.5.1 The Response Message Format

Numeric data are sent in "network byte order" which means that the MSB arrives first. Integers are always 32-bits, other numeric data are 64-bit IEEE floating point values. The raw bytes read for a numeric value must be converted to the machine's byte order before being processed in a program. For integers, the `ntohl` C library function is usually available. For floating values, an example conversion function is provided in the `xclient.cc` file. The byte order is the same as that used by Sun sparc systems, thus this issue can be ignored on those systems, unless code portability is desired.

All response messages begin with a 4-byte integer, which may constitute the entire message in some circumstances. This (and all numeric values) is in network byte order, so must be converted to host byte order before processing. The first integer is the "response code" possibly ORed with the "longform" flag. The response code is an integer 0-9, and the longform flag is hex value 80.

If the longform flag is not set, then no more data exists in the message. Otherwise, most response codes will be followed by additional data. The possible responses are described below.

0

This is the server "ok" message. There is no additional data.

1

This is the server "more" message. There is no additional data. This response is given when the server is waiting for input required to complete a script conditional block, for example:

command	**response**
`keepall`	0
`if (x)`	1
...	
`end`	0

2

This is the server "error" message. There is no additional data. This response is given if the command produces an error.

3

This is the server "scalar" message. If the longform flag is set, there are 8 bytes of following data, representing a double-precision IEEE floating-point value.

4

This is the server "string" message. If the longform flag is set, a 4-byte size integer follows, in turn followed by the string characters. The size value is the number of characters in the string and includes the null termination character of ASCII strings.

5

This is the server "array" message. If the longform flag is set, a 4-byte integer follows, giving the number of elements in the array. This is followed by the array data, 8 bytes per element, in IEEE double-precision floating-point form.

6

This is the server "zlist" message. If the longform flag is set, a 4-byte integer follows, which gives the number of trapezoids in the list. This is followed by the trapezoid list data, with 24 bytes per trapezoid (six 4-byte integers each). The values are coordinates in the internal units (usually 1000 units per micron), in the order *xll*, *xlr*, *yl*, *xul*, *xur*, *yu*.

7

This is the server "lexpr" message, which is the return for the layer expression type. This is treated as a string. If the longform flag is set, a 4-byte size integer follows, followed by the text of the layer expression. The size includes the null termination character of the string.

8

This is the server "handle" message, which is the return for all handle types. This is basically useless on the local machine, since the underlying data resides on the server. If the longform flag is set, a 4-byte integer follows, which gives the handle identification value.

9

This is the server geometry stream message. This message always returns data, the longform flag is ignored. The type 9 return is unique to the geometry stream response from the `geom` command. The ASCII string responses from the `geom` command use type 4 in the normal way, though they are always in "longform". The type 9 record is very similar to a string, however the first 8 bytes of the string contains two integers: the first integer is the compressed size of the following data, and the second integer is the uncompressed size. The compressed size can be zero, in which case compression is not used. The actual string length is the compressed size if nonzero, otherwise the uncompressed size. The string contains OASIS geometry records, as in a CBLOCK if compressed.

The user will have to supply an OASIS reader to interpret the stream. *Xic* provides script functions for this purpose.

4.5.2 Operation

Internal script variables are defined and set in accord with instructions received. The variables and other context are cleared when an initial connection to the server is made or or final connection broken (and `keepall` is not in effect), or when "`reset`" is given.

Other state, such as the current directory and cells in *Xic* memory, is persistent, thus users should initialize *Xic* appropriately, and clear the database before closing the connection.

While in server mode (also in batch mode) the *Xic* functions that query the user for some decision are not available. If the prompt line editor is invoked, it will return immediately as if the user hit **Enter**. The return value is the default string, if any, or any text that was previously supplied with the `StuffText` function. The **Merge Control** behavior is as if the NoAskOverwrite variable was set, i.e., the overwriting behavior will be the default as set with the NoOverwritePhys and NoOverwriteElec variables. If neither of these is set, the action will be to overwrite the cell in memory.

The server produces a log file directory in the same manner as under normal *Xic* operation. These files are removed when the server exits normally, i.e., when a "`kill`" command is received. In server mode, there are files used that are not used in normal mode:

`daemon.log`
: This records connection activity to the daemon.

`daemon_out.log`
: This records the "stdout" channel from the daemon, i.e., the text that would go to the console in normal mode. Under Microsoft Windows, this file is not located with the other log files, but is created in the parent directory of the directory containing the log files. This is due to a technical issue in Windows.

`daemon_err.log`
: This records the "stderr" channel from the daemon, i.e., the error text that would go to the console in normal mode. Under Microsoft Windows, this file is not located with the other log files, but is created in the parent directory of the directory containing the log files. This is due to a technical issue in Windows.

This page intentionally left blank.

Chapter 5

Parameterized Cells and Vias

5.1 Parameterized Cells

Parameterized cells, or "pcells" (or sometimes called "template cells") are cells which in addition to possible fixed geometry, contain an executable program that creates geometry according to one or more parameters supplied to the cell. The cell is instantiated for given sets of parameters, so that instances may have layouts that differ. Parameterized cells are often used to represent devices such as MOSFETS that may come in many shapes and flavors. The MOSFET parameters select the size and other properties of each instantiation. As an alternative, in a process design kit one might find hundreds of fixed-cells with different permutations of size and other parameters. A single parameterized cell that replaces the collection of fixed cells can streamline the design process, provide greater flexibility, and reduce errors.

The full and *XicII* feature sets have support for native and OpenAccess-based portable pcells, as well as the ability to work with the Cadence Virtuoso Express PCells^{TM} feature. The *Xiv* feature set, does not support pcells.

There is an ongoing effort to strengthen the parameterized cell capabilities in *Xic*. The effort includes

- Providing support for languages other than the native script language. In particular, the Python language appears to be the choice for "open" pcells, i.e., pcells which can be used in tools from different vendors.
- Provide commonality and support for Ciranova open pcells and standards.
- Provide commonality and support for the OpenAccess pcell framework.

5.1.1 How PCells Work

Provided below are definitions of some terms used frequently in the discussion that follows.

pcell
: A "parameterized cell" or "template cell". This is a cell containing an executable component, which acts on a set of one or more parameters. When placed in a layout, the cell constructs itself according to the parameters given while instantiating.

super-master
: A pcell in memory.

sub-master
: A master cell created from a super-master and a given parameter set. Instances of the sub-master are actually placed into the layout. A pcell itself is never placed in a layout.

All pcells "work" as follows. The pcell is supplied as a cell file to the design system, which understands the file syntax. Within the design system, an in-memory object called a "super-master" is created, which is an in-memory representation of the pcell. This element contains a list of parameter names, and for each parameter a default value and acceptable range. The element also provides, by some means, a program or script that can be executed from the design system.

When a user wishes to place an instance of a pcell, the pcell is selected from a menu, which causes the pcell file to be read from disk and a super-master created in memory. The user will then specify the parameter values to the cell to instantiate. This is usually done with a pop-up form, where the user can enter values for the various parameters, all of which have defaults. When this entry is complete, the design system will execute the pcell script with the entered parameter values. The result will be creation of a cell in memory containing geometry created by the script in accord with the parameters. This cell is called a "sub-master". It is a normal cell is every respect, though it has properties that link it to the original pcell super-master. Instances of this sub-master are created where the user specifies. A separate sub-master will be created for every differing parameter set that the user provides. Each instance of a sub-master contains properties that contain the parameter set used for instantiation, and the name of the original pcell.

A design containing pcells can be saved in two ways. For a local save, for use in the same design environment, the super- and sub-masters in memory are discarded (or the sub-masters may be cached). When the design is read in again, the instances provide the location of the pcell and the parameter set, which are used to recreate the sub-masters. If instead the design is being sent to another environment, one which perhaps does not handle the pcells, the sub-masters can be written to disk as ordinary cells. The resulting hierarchy will be normal and portable. In *Xic*, sub-masters can be included in saved archive files when the **PCellKeepSubMasters** variable or equivalently the check box in the **Export Control** panel is set, or when the StripForExport variable or equivalent check box in the same panel is set. If a cell is read from a file and is recognized as a pcell sub-master, the PCKEEP cell flag will be set. This will cause the cell to be written to output, whether or not writing of pcell sub-masters is enabled.

5.1.2 PCell History and Status

Historically, the pcell concept was developed for the Cadence Virtuoso layout editor, and supported pcells used the Skill^{TM} language which is the scripting language of the Virtuoso system. This remains the dominant type of pcell around, due to the ubiquity of Cadence installations. However, the Skill language is not available outside of the Cadence environment, so these pcells are not portable to other tools.

The OpenAccess project addressed the pcell portability problem by providing a standardized interface for pcells, with the execution being carried out through a "plug-in" that a vendor, or user, may supply. A pcell, in concept, can be created to use any suitable programming language, provided that the tool used to instantiate the pcell is capable of executing that language. With OpenAccess, the portability problem is reduced to obtaining a plug-in for the pcell language.

There are example plug-ins distributed with OpenAccess that handle Tcl and C++. Unfortunately, the Skill language is not available for general use outside of the Cadence environment. It is not really attractive anyway, as it was developed back in the prehistoric days when Lisp ws "cool", and abominations like EDIF seemed important. There are far better languages, such as Python, available today.

The concept of portable pcells was championed by a company called Ciranova, that supplied an OpenAccess plug-in for Python. They released this, along with companion applications for Python pcell ("PyCell") development, examples, and precompiled OpenAccess and Python libraries as a free "PyCell Studio" download. Ciranova was subsequently bought by Synopsys, but the PyCell Studio remains available and apparently is still under development. An industry group, IPLnow.com (`http://www.iplnow.com`) which includes TSMC and other foundries and some tool vendors, is pushing the cause of "interoperable" PDK libraries based on portable pcells.

Xic is intended to be fully compatible with the PyCell Studio and PyCells, through the OpenAccess interface plug-in. In addition, *Xic* without OpenAccess provides support for Python pcells, and the Ciranova protocols for stretch handles and abutment. However, Ciranova provides a number of library modules and functions as part of its Python implementation that are not present without the Ciranova plug-in and OpenAccess.

Xic with OpenAccess has some limited capability with Skill-based pcells through the Virtuoso Express PCells feature. This allows export is pre-instantiated cached sub-masters of pcells, but not the pcells themselves. This capability is provided through the same OpenAccess plug-in technology mentioned above, but in this case if the parameter set does not have a pre-built sub-master in cache, the instantiation will fail.

The **!rmpcprops** command will remove the properties that make pcells special throughout the hierarchy of the current cell. This operation is not undoable, and renders the hierarchy henceforth free of any pcell history. The user may wish to do this to hierarchies imported from Virtuoso, as the Skill pcells can not be evaluated in the *Xic* environment. In this case, retaining the pcell identities may be pointless, and in fact this may cause trouble, for example when writing output pcell sub-masters are not written unless the user overrides the default (e.g., by checking the box in the **Export Control** panel).

5.1.3 *Xic* Native PCells

Xic supports pcells using the native scripting language, plus Python or Tcl if the respective plug-ins are loaded. Parameterized cells are supported only in physical mode. This section will describe how to create and use native pcells in *Xic*. By "native", execution within *Xic* rather than through OpenAccess is meant. As will be seen, native pcells can be saved in OpenAccess, too, and they are still native.

There are several example native pcells provided in the examples directory of the *Xic* installation. These provide samples of the syntax used in the property strings and other aspects, with comments, and their study should facilitate understanding how to write native pcells.

A native pcell can be saved in any format supported by *Xic*, with certain limitations to be described. Probably, the native cell format is the most convenient. These can be easily edited with a text editor, which the advanced developer is likely to do on occasion.

A pcell can have any name that is compatible with *Xic*. Earlier releases of *Xic* required that a pcell name have a literal "`XXX`" suffix. This is no longer the case, but if the `XXX` is present, it will be stripped in sub-master names and replaced with a unique identifying code for the parameter set. Otherwise, the code is appended to the pcell name.

Super- and sub-master cells, and sub-master instances, differ from normal cells and instances by the presence of a few special properties. These are:

pc_name property, number 7197

This property is assigned by *Xic* to pcell sub-masters and their instances. It provides the name of the pcell from which the sub-master or instance was derived.

pc_params property, number 7198

This property is assigned by the user to pcells, and contains the default parameter set. It will be assigned by *Xic* to sub-masters and instances, and contains the parameter set that was used to create the sub-master.

The string of the pc_params property has the form

[*typechar*:]*name*[=]*value*[:*constraint*] [[,] ...]

The string consists of a series of *name* and *value* tokens. The *name*s can not contain white space or punctuation. Ahead of the *name* is a type specification character if the value is not string type. In native pcells, all parameters are (for this purpose) string type, so the type specifier will never appear. However, the syntax used may be extended in future, so it is documented in the table below. All types except for string type will have a specifier. These **will** appear in property strings obtained from OpenAccess for non-native pcells.

`b`	boolean
`i`	integer
`t`	time value
`f`	32-bit float
`d`	64-bit float

Each of the *name* tokens is the name of a parameter that can be applied to the pcell. These will become names of variables in the script, so that these names should not be defined or used in the script text in a conflicting way.

Every *name* should have a *value*, an "empty" value is specified as an empty string (`""`). The *value* is separated from the *name* by white space, a comma, or an equal sign. The *values* are taken as default values for the parameters, and can be numeric values or strings. A *value* that contains white space, commas, or colons should be quoted. The value string can also be an executable code fragment using only parameters already defined (to the left) and constants, for example

```
param1=2,param2="param1 + 1"
```

This form, however, can not be used with constraints (see below). It can also only appear in super-master pc_params properties. the pc_params strings of sub-masters and instances must have constant values.

The quoting behavior is a bit complicated, so as to support Python and native languages. If the *value* is quoted with double-quote marks, the double quote marks will be stripped, and the parameter will take the enclosed characters. However, if a backslash character ('\') appears ahead of the first double quote, the double quote marks will be retained. In the native language, this will ensure that the parameter is string-type.

For example

```
myvar="123"
```

The parameter (variable) `myvar` will be assigned the value 123, causing it to become scalar-type. On the other hand

```
myvar=\"123"
```

will assign "123" to myvar (including the quotes) thus myvar will be string-type. In general, if the *value* is to be taken as a string constant in the native language, a backslash should be placed ahead of the first double quote mark.

If the *value* is quoted with single-quote marks, the single-quote marks are retained, along with the characters between them. This is for Python support. However, if the second character is a double-quote mark, the single-quote marks will be stripped, leaving the double-quoted result. This is an alternative and somewhat deprecated way to specify a string constant in the native language.

```
mystring='"a string constant"'
```

In any case, when the parser is searching for the ending quote mark (single or double), if the mark is found but it is preceded by a backslash, both characters are taken verbatim and the search continues. Thus, the backslash can be used to hide quote marks of the same type in the string.

If the *value* is a constant (not an executable fragment), the *value* can be followed by an optional constraint specification, separated from the *value* by a colon (no white space is allowed around the colon). Constraints define the scceptable values for the parameter, using a syntax described in 5.2. The constraints appear only in pc_params properties of super-masters, and are not copied to pc_params properties of sub-masters and instances.

The parameter string is logically converted to a series of assignment statements which are executed before the script. For example, the parameter string

```
param1=1.0,name=\"my template",param2="param1 * 2"
```

would map to the following logical script lines

```
param1 = 1.0
name = "my template"
param2 = param1 * 2
```

pc_script property, number 7199

This property is assigned by the user to a pcell, and appears only in the super-master. It contains the script, or a path to a script, which is executed when the pcell is instantiated.

The pc_script property text is in the form

[@LANG *langtok*] @READ *path* [@MD5 *digest*] | *script text*

The @LANG, @READ, and @MD5 tokens are literal. The *langtok* may be one of (case insensitive)

n[ative]	native sript, the default
p[ython]	python script
t[cl]	tcl script

The *path* token must appear if @READ is given. If @READ is not given, any remaining text is taken as literal executable script text.

The *path* is to a file containing the executable text, and should be quoted if it contains white space. If the *path* is not rooted, it will be searched for in a directory search path set in the PCellScriptPath variable.

When a path is given, one can also apply the @MD5 *digest* clause. The *digest* is that for the script file, and can be obtained from the **!md5** command, or the Md5Digest script function, or from the command

openssl dgst -md5 *filepath*

on most Linux systems. If given, the script file digest must match the digest given, or the script will not be executed. This will ensure that only the "correct" script file is used.

Previous versions of *Xic* required that the script actually appear in the pc_script property string. This can still be done, and may be convenient for many pcells, particularly very simple ones. However, one may encounter a portability issue caused by string length limitations of the GDSII and CGX formats due to their maximum record length of 64KB. The native cell format, the CIF format as extended by Whiteley Research, and the OASIS format have no built-in string length limit, nor does OpenAccess.

When using separate script files, for portability it may be best to **not** provide a full path to the script in the pc_script property string, but give the file name only and use the search path variable. Then, the scripts can be kept in different locations at different *Xic* installations, and pcells will still be portable provided the PCellScriptPath is set (probably from an initialization file). The `MD5` digest keying can ensure that the script file found via the search path is correct, or it will not execute.

The script, whether in a separate file or not, is basically conventional, and uses the native object creation functions to build up the geometry, presumably using the parameter values as input. The example native pcells provided with the *Xic* distributions in the examples directory illustrate how the script is incorporated.

One aspect of importance is the script return value, which will tell the calling program whether or not script evaluation succeeded. If evaluation fails, *Xic* can gracefully "clean up" by destroying partially completed sub-masters, and any corresponding instance placements, and alerting the user to the error.

The script should return 0 (zero) on success, which is the default if no explicit return value is specified. Any nonzero return value indicates faliure. The mechanics of setting the return value differs between the supported languages, and is described below. In every case, just before a nonzero value is returned, the `AddError` function should be called with a message explaining the error.

Native

The `return` keyword, followed by a value, will terminate the scrpt and return the value. For example, here is a snippet that checks the value of a parameter named "`top`" and fails if it is out of range:

```
if (top < 1 | top > 20)
   AddError("Parameter top is out of range [1 - 20].")
   return 1
end
```

Actually, if the value following `return` is omitted, the return value is 1, so just a bare "`return`" will signal the error condition. If the end of execution is reached and no `return` keyword is encountered, the value returned is 0 (success). If the script is terminated with the `Halt` or `Exit` functions, the return value is 0. If the script is halted by an internal error, the return value is -1. If the script is halted due to an interrupt signal, the return value is 1.

Python

The recommended way to induce an error exit in a Python script is to call "`sys.exit`" with a nonzero argument. The example above translated to Python will read:

```
if (top < 1 or top > 20):
   xic.AddError("Parameter top is out of range [1 - 20].")
   sys.exit(1)
```

Errors detected by the Python interpreter are passed back as nonzero exit returns.

Tcl

The recommended way to induce an error exit from a Tcl script is to call "`return -code error`". The example above translated to Tcl will read:

```
if {$top < 1 || $top > 20} {
   AddError {"Parameter top is out of range [1 - 20]."}
   return -code error
}
```

Errors detected by the Tcl interpreter are passed back as nonzero exit returns.

To summarize, a pcell is never itself instantiated. When one places an instance of a pcell, the following steps occur:

1. The pcell is read into memory as a "super-master" if it is not already there.
2. The user enters the parameter values.
3. The database is searched for another cell derived from the same pcell with the same parameter values, i.e., an equivalent sub-master. If one is found, a new instance is created and given pc_name and pc_params properties copied from the sub-master, and we're done.
4. Otherwise, the script is executed, in the context of a new, empty cell whose name consists of the pcell name suffixed by a unique identifier. This is the sub-master cell. It is given a pc_name property to identify the pcell, and a pc_params property to list the parameters used. The new sub-master is instantiated and the instance given the same two properties, and we're done.

Once the instance is placed, it behaves in all respects as a normal cell. It has a "master" derived from the pcell as a sub-master, and a unique sub-master exists for each unique parameter set. Writing the hierarchy, including the sub-masters, to an archive produces a perfectly normal file. However, by default the sub-masters are **not** written to output, instead they are expected to be recreated from the pcell when needed. The pcells (super-masters) are **never** included in the output file, since they are not directly instantiated in the hierarchy. Thus, when exporting, the pcell should be supplied separately, if needed. If sub-masters are included in the archive, then the pcell is not needed, unless further parameter changes are required. In *Xic*, sub-masters can be included in saved archive files when the PCellKeepSubMasters variable or equivalently the check box in the **Export Control** panel is set, or when the StripForExport variable or equivalent check box in the same panel is set.

5.1.4 Creation of a Native Parameterized Cell

To create a native pcell, one can follow this procedure:

Write the script

Write a script that creates the geometry desired, in response to a set of variables that will become the parameters. The script can be authored as any other script. It should be thoroughly debugged before committing it to a parameterized cell.

It is recommended that the top of the script contain a comment listing the parameters and their purposes, and explicit tests of the values that will abort the script (returning nonzero) if a value is out of range or otherwise not acceptable. Any nonzero return should have a call to `AddError` explaining the error. This text will be included in the system error reporting.

Create the parameterized cell
Use the **Open** (**File menu**) command to edit a new cell which will become the pcell. Add any fixed geometry to the cell that is necessary. This can be done at any time. Keep this cell as the current cell and add the properties listed below.

Add the pc_script property
Bring up the **Cell Property Editor** (**Edit Menu**). Press **Add**, which brings up a pop-up menu, and select **pc_script** in the pop-up menu. This will prompt for the property string on the prompt line.

At this point we need to decide whether to incorporate the script into the property string itself, or to keep the script in a separate file. One consideration is that GDSII and CGX files have 16-bit record lengths, which will limit the lengths of property strings. In the present *Xic* release, CIF and native string lengths, and OASIS string lengths, are unlimited. There is also no limit when storing the cell in OpenAccess.

First, assume that the script is to be stored in the property string. We will use the "long text" feature to facilitate entering the script.

Enter property text with script
Press the "**L**" button to the left of the prompt line. This brings up the **Text Editor** pop-up. If the script text is Python or Tcl, a `@LANG` specification must appear first. Type one of the following into the editor window. For Python

```
@LANG Python
```

or if Tcl

```
@LANG Tcl
```

Neither is needed for native script language.

The next step is to import the script text. This is presumed to exist in a file, though for very simple scripts an advanced user can type it in. For the script in a file, one can use the **Read** button of the text editor (in the **File** menu) to read in the script file. Then perform any last minute editing, such as removal of the variable declarations that would be redundant with the parameters.

Press the **Save** button in the **File** menu of the text editor. The text editor will disappear, and the script will have been saved in the pc_script property of the current cell.

Enter property text without script
One can use the "long text" text editor feature, or simply type into the prompt line. Without the script, there generally isn't much to type.

First, if the script text is Python or Tcl, one must enter a `@LANG` specifier as explained above. If needed, just type in the two tokens. Next, enter a `@READ` directive in the form

`@READ` *path*

where *path* is a path to the file containing the script. This can be an absolute path, however it may be more convenient to just specify the file name, and set the PCellScriptPath variable to a directory where pcell script files are kept. Then, the location can change without one having to edit the property string. This completes text entry. Exit the text editor as above if it is being used, or press **Enter** to terminate text entry into the prompt line. The text is saved in the pc_script property of the current cell.

Optionally, one can append a directive of the form

`@MD5` *digest*

The *digest* is the 32-character string obtained from the **!md5** command for the script file. When included, the script will not execute unless the script file has a matching MD5 digest, which ensures that the script file accessed is the correct one and hasn't been modified.

Add a pc_params property
Next, we program the pcell's parameters and default values by adding a pc_params property. In the **Cell Property Editor** (**Edit Menu**), press **Add**, then select **pc_params** in the pop-up menu.

Again, one can use the "long text" editor, or type directly into the prompt line. For long parameter lists, the editor would be preferred. Enter the parameter list in the format described for this property string (see 5.1.3). If using the editor, any combination of multiple lines and/or multiple specifications per line can be used. A parameter specification consists of a parameter name followed by '=' and its value, optionally followed by a colon and a constraint string (see 5.2). There must be no white space around the colon that delimits the constraint string, but the constraint string itself may contain white space, which is ignored.

Save the text if using the text editor, or press **Enter** if using the prompt line, when done.

Add additional properties
There are other properties that may be required, to support stretch handles (draggable edges, see 5.4) and auto-abutment (see 5.5) protocols. Text is added as for the properties we've described. This may be a second pass, after getting the basic cell working.

Save the current cell to disk, the native format is probably most convenient. Congratulations, you have yourself a pcell!

5.1.5 Adding an Instance of a Parameterized Cell

Adding a pcell to the current layout is the same procedure, whether the pcell is native, or not. One adds an instance of a pcell like one would add an instance of any other cell. If a native pcell, the cell file name can be given to the **New** text entry pop-up of the **Cell Placement Control** panel brought up with the **place** button in the side menu.

Pcells saved in OpenAccess can be instantiated with the **Place** button in the **Contents** listing window from the **OpenAccess Libraries** pop-up from the **File Menu**. These cells are also available through the **Cell Placement Control** panel. In the text input pop-up from the **New** button, enter the OpenAccess library name that contains the desired pcell, followed by space, then the pcell name.

When cell placement becomes active, by pressing the **Place** button the **Cell Placement Control** panel, the **Parameters** pop-up appears. This pop-up displays a text entry area for every parameter, loaded with the default value. The user can enter the values desired.

In addition, a double-line box is ghost-drawn and attached to the mouse pointer. This figure does **not** represent the actual size of the instance, in fact it illustrates that the instance size is unknown. The instance size will not be known until the parameter set is used to create or identify the corresponding sub-master cell. This will happen when the user clicks in the drawing window to place an instance. Better, the **Apply** button in the **Parameters** pop-up can be pressed, which will create a sub-master without instance placement. The box attached to the mouse pointer will now be formed with a single line, and will have the actual size.

As with a normal cell, instances are placed where the user clicks. Note that the **Parameters** pop-up remains visible while instances are being placed. The parameters can be changed, and the **Apply** button pressed, to change the type of instantiation to be subsequently placed. Note that the subsequent instances will use the new parameter values, pressing **Apply** merely updates the bounding box attached to the mouse pointer.

5.1.6 Changing the Parameters of an Instance

Once a pcell has been instantiated, the instance can be changed to represent a new set of parameter values **if** the pcell is available. Thus, when a design is exported to another site that may wish to modify the cell parameters, the pcells must be exported as well. The pcells are **not** automatically added to GDSII files or the other file formats. They can be supplied as *Xic* cells, in addition to the GDSII or other output. Further, *Xic* native pcells are **not** directly portable to other design systems, they are known to *Xic* only.

One possible way to maintain native pcells is to place them in a library.

Assuming that the pcell is available, one can change the parameters of an existing pcell instance with the following procedure. First, select the pcell instance to modify. Then, while holding down the **Ctrl** key, click on the selected pcell. The **Parameters** panel will appear. One can now change parameter values as needed, and press **Apply** to reparameterize the instance.

Less conveniently, the pc_params property can be edited with the **Property Editor** with the same effect. Bring up the **Property Editor** with the **Properties** button in the **Edit Menu**. With the editor active, click on a pcell instance. The instance will be marked, and its properties listed. Among the listed properties will be the pc_params. Click on this entry in the listing window, the text will show as selected. Then, press the **Edit** button in the **Property Editor**, which will bring up our old friend the **Parameters** panel. Adjust the parameters, then press **Apply**. The new parameter set will be applied to the marked instance.

5.1.7 Changing the Parameters of a Sub-Master

One can change all of the instances that use a particular parameter set to a new parameter set by changing the parameters of the sub-master cell of the instances. The original pcell must be accessible, as for changing individual instances. The procedure is to edit the parameters of a sub-master, which will have the effect of reparameterizing all of its instances.

A quick way to do this is to select an instance of the sub-master to be edited, and press the **Push** button in the **Cells Menu**. The editing context will be pushed to the sub-master. The sub-master can also be selected for editing from the **Cells Listing** pop-up (**Cells Menu**), or by giving its name in the **Open** command (**File menu**).

With the sub-master as the current cell, bring up the **Cell Property Editor** with the **Cell Properties** button in the **Edit Menu**. The listing of properties will include a line for the pc_params property. Select the property by clicking on it, then press the **Edit** button. Again, the **Parameters** pop-up will appear. One should modify the parameters desired, then press **Apply**. The new parameter set will then apply to the instance pushed into, and all other instances of the same sub-master. Use the **Pop** button in the **Cells Menu** to return to the original editing context if **Push** was used.

5.2 Parameter Constraints

Constraints are described by text strings included in the pc_params property contained in the super-master cell. Constraints do not appear in the sub-master or instance properties. Constraint support is also provided for Ciranove/Python OpenAccess pcells, though the constraint strings are provided by another method internally as there are no corresponding super-master *Xic* cells.

In *Xic*, constraints are mainly handled in the **Parameters** panel (see 5.3), which is where parameter setting is primarily handled. The constraints may affect the type of input widget for the parameter. It will not be possible to set a value for the parameter that is not allowed by the associated constraint.

The constraint strings follow closely the Ciranova format. Each is in the form of a Python function call, with a set of arguments that define the constraint. The arguments can be either positional or named. For example, the `range` constraint has the following template:

`range(`*low*,*high*,*resolution*=`None`,*action*=`REJECT)`

The two final arguments have defaults, and are therefor optional. Arguments can be given positionally, or as an assignment using the argument name keyword. The following forms are equivalent:

```
range(0,10)
range(high=10,low=0)
```

The first line follows the argument order of the template. The second line does not, but supplies the argument name explicitly. Arguments can appear in any order if the name is given. An argument list can use both positional and explicit assignment. Note that the *resolution* and *action* arguments are not given in either example, so that the defaults will be used.

All keywords are case-insensitive.

Each constraint type contains an *action* argument, which can be set to one of the literal enumerators `REJECT`, `ACCEPT`, or `USE_DEFAULT`. This specifies what happens when an attempt is made to set the parameter to a value not allowed by the constraint. The `REJECT` option (the default) will simply fail, causing the command that initiated the operation to also fail. The `ACCEPT` action will accept the new parameter value, basically ignoring the constraint. The `USE_DEFAULT` option is intended to reset the parameter to the default value when the constraint test fails, but this is not implemented in *Xic*, `REJECT` will be done instead.

The enumeration value `None` can be given to most arguments. This usually means to ignore the argument, and skip any test that would use the argument. For example, a range constraint may give a *high* value of `None`, meaning that the parameter value can be arbitrarily large.

The available constraint types are as follows.

`choice`

The `choice` constraint restricts the parameter to a number of alternatives. These alternatives can be numbers or strings, as appropriate for the parameter data type. The keyword "`choiceConstraint`" is a (case-insensitive) synonym. The template is

`choice(`*choices*,*action*=`REJECT)`

where the *choices* argument is a list in the form

`[`*element*,*element*...`]`

The square brackets are literal, *elements* are numbers or strings (single or double-quoted) which are separated by commas.

Examples:
`choice([1,2,4,8])`
`choice(["red","green","blue"])`

The first line restricts the numeric parameter to the values listed. The second line would restrict a string parameter to the strings listed. Note that if the script is Python, single quotes must be used instead of double quotes. Single or double quotes can be used with native scripts.

`range`

The `ramge` constraint restricts a numerical parameter to a range of values. The keyword "`rangeConstraint`" (case-insensitive) is a synonym. The template is

`range(`*low*,*high*,*resolution*=`None`,*action*=`REJECT)`

The *low* and *high* are numerical endpoints of the range. Either can be the enumeration value `None`, which skips testing against that endpoint. For example,

`range(0,None)`

simply indicates that the value must be zero or larger.

The numerical values passed for *low* and *high* must be consistent with the language used for the script. In particular, Python requires a standard integer or floating-point format. The native language allows SPICE-type numbers (e.g., `1.2K`), hex numbers with a "`0x`" prefix (e.g., `0xff00`) and character constants (e.g., `'\n'`) in addition.

The *resolution* argument is used in the **Parameters** panel to set the number of digits to include following a decimal point (see 5.3).

`step`

The `step` constraint limits the numerical parameter value to multiples of a given delta between a starting and ending value. The keyword "`stepConstraint`" (case-insensitive) is a synonym. The template is

`step(`*step*,*start*=`0`,*limit*=`None`,*resolution*=`None`,*action*=`REJECT)`

The parameter must be numeric. If the *step* value is 0 or `None`, the constraint acts the same as the `range` constraint, with *start* and *limit* providing the low and high values, respectively.

Otherwise, the allowed values are given by

start `+ N*`*step*

where `N` is a non-negative integer, and the value of the expression is within the range terminated by *limit*, if *limit* is not `None`. Note that *step* can be negative, in which case the parameter value must be greater than or equal to *limit*.

The *resolution* is treated as in the `range` constraint.

`numericStep`

This is very much like the `step` constraint, but is intended for use with string variables used for numeric input to support SPICE-like multipliers. This is needed for script languages that don't handle numbers in this format. Since the native script language understands this number format directly, it is not clear that the `numericStep` constraint will ever be needed in pcells with native scripts. The keyword "`numericStepConstraint`" is a synonym. The template is

`step(`*step*,*start*=`0`,*limit*=`None`,*resolution*=`None`,*scaleFactor*=`'u'`,*action*=`REJECT)`

The arguments are the same as for the `step` constraint, with the addition of *scaleFactor*. The *scaleFactor* is a string set to one of the scaling suffixes from the table below:

suffix	multiplier	name
a	1e-18	atto
f	1e-15	femto
p	1e-12	pico
n	1e-9	nano
u	1e-6	micro
m	1e-3	milli
mil	25.4	mil
k	1e3	kilo
meg	1e6	mega
g	1e9	giga
t	1e12	tera

The scale fastor is case-insensitive. If the *scaleFactor* is assigned the value `None`, no scale fastor is assumed, and the constraint is basically identical to `step`. If a scale factor is given, numbers given for *step*, *start*, and *limit* are internally multiplied by the scale factor, before comparison to the parameter value.

5.3 Parameters Panel: Set PCell Parameters

The **Parameters** panel appears when it is necessary to provide parameters for a parameterized cell (pcell) instantiation. These situations include

- During placement of pcell instances with the **Cell Placement Control** panel from the **place** button in the side menu.
- While editing a pc_params instance property with the **Property Editor**, which is obtained with the **Properties** button in the **Edit Menu**.
- If the user clicks with button 1 and the **Ctrl** key held on a selected pcell instance, The **Parameters** panel will appear. The user can reparameterize the instance.
- While editing the pc_params property of the current cell with the **Cell Property Editor**, which is obtained with the **Cell Properties** button in the **Edit Menu**.
- If one opens a non-native pcell for editing, the **Parameters** panel will appear. In this case, the label on the leftmost button is "**Open**" rather than "**Apply**". Entering parameters then pressing **Open** will create or find the sub-master for the parameter set, and make it the current cell. This will not happen with native pcells, which can be edited directly in *Xic*.

The **Parameters** panel provides an entry area for each pcell parameter. In cases where there more parameters than will fit within the window, a scroll bar will appear, allowing the user to scroll the parameter listing. The listing order of the parameters is as provided by the pcell.

The type of entry widget shown in the panel depends on the data type of the parameter, and the parameter constraint specification. The constraint string, if any, is obtained from the pc_params property of the pcell super-master. The following logic is used:

- If the parameter is boolean, any constraint is ignored, and a check box is created.
- If the parameter has a `choice` constraint, a drop-down menu containing the given choices is created. The choices can be numeric or string values.
- If the parameter has a `range` or `step` constraint, a numeric entry "spin" button is created. The numbers displayed in the text area follow the constraint, i.e., the range is limited, and the step value (if any) is enforced. The up/down arrows add or subtract a step value. Further, the floating-point precision used for the number will follow the *resolution* value of the constraint. This is described below.
- If the parameter has a `numericStep` constraint, the set-up is very similar to the `step` constraint, but an additional label will appear showing the *scaleFactor*, if any. This scale factor is logically appeaded to the number that appears in the entry area.
- If there is no constraint, a simple text-entry area is created.

For numeric entries, the constraint *resolution* value will set the number of digits that follow the decimal point in the display. For the default value of `None`, or if less than 1.0, the number of digits will be based on the current database resolution, as set at program startup with the DatabaseResolution variable. If the resolution is the default value of 1000, three digits will be used (`1.235`), otherwise four (`1.2345`).

Otherwise, the number of digits following the decimal will be set by the following logic:

```
if (resol > 1e5) num = 6
else if (resol > 1e4) num = 5
else if (resol > 1e3) num = 4
else if (resol > 1e2) num = 3
else if (resol > 1e1) num = 2
else if (resol > 1e0) num = 1
else num = 0
```

Note that giving a *resolution* of 1.0 will set the number of digits to zero, indicating integer values only (no decimal point is shown in this case).

The panel logic differs somewhat depending on the context. When editing an existing property, with the **Property Editor** or **Cell Property Editor**, the **Parameters** panel is "modal", meaning that the rest of *Xic* is inactive while the panel is visible. The user is expected to enter the appropriate parameter data and either press **Apply** which will accept the new parameter set, or **Dismiss**, which will abort the current parameter edit. In both cases, the **Parameters** panel will disappear, and *Xic* will return to normal status. The **Reset** button will revert all parameter settings in the panel to the initial settings when the panel was created, i.e., the values from the existing property string.

When placing instances, on the other hand, the **Parameters** panel is not modal. The parameters can be changed at any time, and the changed parameter set will apply when new instances are created, whether or not **Apply** is pressed. Pressing **Apply** will create or find the existing sub-master for the parameter set, from which the instance bounding box is obtained and used in the ghost-highlighting during instance creation. The **Dismiss** button will remove the panel, but the instance placement will continue. The **Reset** button will reset all parameter values displayed in the panel to the defaults provided in the pcell.

When opening a foreign pcell, the **Parameters** panel is non-model, and nothing happens unless/until **Open** is pressed. Pressing **Open** will create a new sub-master if necessary for the parameters as set, and make the sub-master the current cell for editing. Editing the sub-master is generally not a great idea, unless the user understands the issues. Changing the pc_params property, though, is a valid way to modify all instances of the master. Other changes to the sub-master will be lost, unless the sub-master is saved, possibly with the PCellKeepSubMasters variable set. Pressing **Dismiss** simply retires the panel. Pressing **Reset** returns all parameter values shown in the panel to the pcell default values.

5.4 PCell Stretch Handles

Xic supports the protocol for stretch handles defined by Ciranova. This provides support for stretch handles defined in PyCells, but also allows use of stretch handles in native pcells.

A stretch handle is a graphical item that can be moved with the mouse pointer, where the motion causes a change in a parameter value. Usually, the object is associated with a parameterized cell instantiation, and motion causes remastering of the instance to a new sub-master created with the new parameter. For example, stretch handles might be used to graphically change the gate length and width of a MOSFET pcell instance, if the corresponding pcell supports the protocol.

Stretch handles are visible and activated only when the containing instance is shown large enough on-screen, to avoid false-triggering. The size threshold can be set from the **PCell Control** panel from the **Edit Menu**, or equivalently with the PCellGripInstSize variable.

In *Xic*, when editing a sub-master containing stretch handles, the handles are visible as well, and can be moved. This will change the parameterization of the sub-master, and all of its instances. This is equivalent to modifying the pc_params property with the **Cell Property Editor** from the **Edit Menu**.

If the **Hide and disable stretch handles** check box in the **PCell Control** panel from the **Edit Menu** is checked, or equivalently if the PCellHideGrips variable is set, all stretch handles will be invisible and disabled.

Adding stretch handles to a pcell amounts to adding box objects with the grip property applied. The grip property provides the setup information.

There are example capacitor pcells that use stretch handles that can be found in the examples directory of the *Xic* program distribution area. These demonstrate use of stretch handles and illustrate the property syntax.

grip property, number 7195

This property is very similar to the Ciranova pycStretch property, used to implement stretch handles. The property has meaning when applied to physical-mode boxes only. The property string has the following format:

`name:`*val*; `stretchType:`*val*, `direction:`*val*, `parameter:`*val*, `minVal:`*val*, `maxVal:`*val*, `location:`*val*, `userScale:`*val*, `userSnap:`*val*, `key:`*val*

The terms have precisely the same names and interpretation as the pycStretch property described in the *Ciranova PyCell EDA Tool Integration Guidelines* document provided with the Ciranova PyCell Studio package (now available from Synopsys). However, there are some differences.

1. Ciranova does not allow white space within the string. In *Xic*, white space can appear between the terms as shown above.
2. The semicolon following the name and the commas are optional, the terms can be white-space separated.

3. In both cases a property string can contain multiple grip specifications. Ciranova separates the specifications by white space. In *Xic*, a new specification is started whenever a keyword is repeated.
4. Ciranova requires that all keywords be provided in each specification, except for the name, which can be omitted for names with varying `key` strings. In *Xic*, when parsing multiple specifications, previous values of the various parameters are retained, so only changed values need be given.
5. *Xic* keyword matching is case-insensitive.

The terms have the following significance.

`name`
A name for the stretch handle, which should be a unique string token within the pcell.

`stretchType`
Set to one of the keywords '`relative`" or "`absolute`". Per Ciranova, if `relative`, the increment is measured relative to the center of the rectangle, while `absolute` is the increment measured according to the absolute X and Y directions. This parameter is ignored in *Xic*, since the explanation does not seem to make sense.

`direction`
Set to one of the keywords "`NORTH_SOUTH`" or "`EAST_WEST`", specifying the translation direction of the stretch handle.

`parameter`
The name of the pcell parameter that is modified by the stretch handle.

`minVal`
A numerical value giving the minimum value of the parameter being modified. SPICE-style scaling suffix values and units, e.g., `1K`, `100nM`, are acceptable, units are ignored.

`maxVal`
A numerical value giving the maximum value of the parameter being modified.

`location`
This specifies the location point for the graphical stretch handle on the layout rectangle. The value must be one of

"`Location.CENTER_LEFT`",
"`Location.LOWER_CENTER`",
"`Location.CENTER_RIGHT`",
"`Location.UPPER_CENTER`",

which specify the left, bottom, right, and top sides. All Ciranova codes are handled, those listed above display a line stretch handle, others will show a glyph.

`userScale`
This is a real number scale factor used to multiply the change in parameter value.

`userSnap`
The real number resolution value which should be used for snapping the parameter value, i.e., the reported parameter value will be an integer multiple of the `userSnap`.

`key`
The name used as a key to specify values for multi-valued parameters, and should be "`None`" for ordinary parameters. Multi-valued parameters are not supported in *Xic*.

In *Xic*, stretch handles are available only in physical mode. They are visible in selected, expanded instances only. A stretch handle is represented as a double-line highlighting of one of the four edges of the rectangle to which the rip property is applied.

The user can drag the highlighted edge in a direction normal to the edge over a range set in the property. The edge is ghost-drawn and attached to the mouse pointer during the move. Unlike some other move operations in *Xic*, only dragging is allowed, clicking on a grip will do nothing special. If the associated parameter has a constraint string defined, the highlighting will be visible only for allowed values of the parameter.

5.5 PCell Abutment

Auto-abutment is most commonly used in MOS transistor pcells. If one overlays two compatible transistor instances, the two instances reconfigure themselves into a dual-gate configuration, eliminating redundant geometry.

At this time, the only available example pcell that implements auto-abutment is the Nmos2 pcell in the IPL_cni130 library supplied with the Synopsys (Ciranova) PyCell Studio download. This is an OpenAccess Python portable pcell which is part if the IPL (IPLnow.com) library of open-source portable pcells.

The following procedure illustrates auto-abutment.

1. Download and install the Synopsys PyCell Studio package. This is free from Synopsys, but requires registration and a password mailback. Versions are available for Linux and Windows, though the Windows version is not currently supported in *Xic*.
2. Start *Xic* in an environment that will load the OpenAccess libraries and Python from the PyStudio. Use "`-Tcni`" to reference the appropriate technology file. Edit an empty cell.
3. Select the **OpenAccess Libs** button in the **File Menu**, which will bring up the libraries list.
4. Select the `IPL_cni130` library by clicking on the name. Then press the **Contents** button. A new listing window will appear.
5. Scroll down in the new window and click on the `Nmos2` entry.
6. Then click the **Place** button in the bottom-right corner of the same window. The **Cell Placement Control** panel will appear. Press the **Place** button in this panel.
7. The **Parameters** panel will appear, and the cell placement icon will be attached to the mouse pointer. Click twice in a drawing window to place two instances of the cell, far enough apart that they don't overlap. Press **Esc** to exit placement mode.
8. Use the **Expand** feature from the **View Menu** to set the display depth so that the instance content will be shown.
9. Now for the fun part. Pop down any pop-up windows or otherwise move them out of the way. Select one of the cell instances, and move it over the other, so that the right contact area of one touches the left contact area of the other. Both instances will reconfigure themselves, and the overlapped contact will be gone! The structure represents a dual-gate transistor.
10. Move one of the instances well away from the other. Note that they revert to their original form.

11. Click the **PCell Control** button in the **Edit Menu**. In the panel that appears, select **Mode 2 (with contact)** for **Auto-abutment mode**.

12. As before, move one of the instances so that the contacts overlap. In this case, note that one of the instances retains the contact. This mode inplements trasistors with a shared contact.

The abutment protocol adheres as closely as possible to the description from the `eda_tool_integration.pdf` document supplied with the PyCell Studio. There is one very significant difference, in that Synopsys incorporates the logic into a separate non-visual pcell, which is created transiently to handle abutment events. In *Xic*, the logic is built into the program. Thus, auto-abutment can be used in native language and Tcl pcells in *Xic*, as well as Python pcells. In *Xic*, the internal logic handles abutment events, the separate pcell is not used.

Auto-abutment is enabled in a cell through application of a number of object properties that define aspects of the abutment. These are applied to objects created in the sub-master (or inherited from the super-master). The *Xic* properties as described below correspond to the properties described for abutment in PyCells, with generally identical syntax.

ab_class
: This is equivalent to the Ciranova pycAbutClass property. It is applied to pin shapes to specify that two pin shapes from different cells can be abutted. Only pins with the same ab_class property string can trigger auto-abutment.

ab_rules
: This is equivalent to the Ciranova pycAbutRules property. The property is applied to each pin shape that can be abutted, and the string specifies how the pcell parameters are modified for different abutment modes.

ab_directs
: This is equivalent to the Ciranova pycAbutDirects property. The property is applied to each pin shape that can be abutted, and the string contains a comma-separated list of one or more of the string tokens `left`, `bottom`, `right`, and `top`. These specify the valid abutment directions.

ab_shapename
: This is equivalent to the Ciranova pycAbutShapeName property. This property is assigned by the pcell developer to each pin shape which can be abutted. It assigns a unique name to the shape.

ab_pinsize
: This is equivalent to the Ciranova pycAbutPinSize The property is applied to each pin shape which can be abutted, and supplies an orientation-independent width parameter.

ab_inst
: This property is applied to instances of abutable cells, and contains an instance name. *Xic* normally does not generate or use instance names.

ab_prior
: This property of a pcell instance indicates that the instance is abutted, and this property contains pre-abutment parameter values for use in reverting abutment.

ab_copy
: This property is applied to instances with ab_prior properties that have just been copied. This will allow parameter reversion of the copy without touching the partner of the original.

5.6 Synopsys (Ciranova) PyCell Studio

Most parameterized cells (pcells) have been written in the Cadence Virtuoso environment, using the proprietary Skill scripting language found only in that environment. These pcells can only be used in a Virtuoso environment.

Ciranova, Inc., now owned by Synopsys, developed and championed the idea of portable pcells, pcells that would have published interfaces and use a common programming language, that could work in any design environment. The company provides a free downloadable "PyCell Studio" design kit. The concept is made possible by the use of OpenAccess, which has a well-defined framework for pcell support, is well documented, and source code is published. Cadence Virtuoso and most modern tools use OpenAccess.

Though OpenAccess provides support for pcell interfacing and management, actual execution of the pcell script is exported to external code supplied as a plug-in. The plug-in provides an interface to the language interpreter or compiler and other things required to execute the script. This plug-in is supplied by the system vendor or user. For example, in a Virtuoso installation, a Skill plug-in is provided. OpenAccess comes with example plug-ins for Tcl and C++.

Ciranova developed a Python plug-in for OpenAccess, with a set of interface functions for creating geometry and related purposes within OpenAccess. Python is a very popular, modern, open source scripting language. It is present on any standard Linux system, and is available for most other operating systems. Ciranova calls portable Python-based pcells that use the Ciranova plug-in "PyCells".

The PyCell Studio design kit contains tools for viewing, testing, and creating PyCells. An example library of PyCells is provided, complete with technology and display resource files. It also provides OpenAccess and Python, so the package is quite complete. There is comprehensive documentation and tutorials.

Though Ciranova has been bought by Synopsys, the PyCell Studio remains available and apparently is still under development. An industry group, IPLnow.com, which includes TSMC and other foundries and some tool vendors, is pushing the cause of "interoperable" PDK libraries based on portable pcells.

Whiteley Research fully supports this effort, and *Xic* will be interoperable with the PyCell Studio design kit and PyCells as much as possible.

5.6.1 Connecting to PyCell Studio

This section describes how *Xic* can directly interface to the PyCell Studio example library and technology. PyCells from the library, or authored by the user, can be instantiated in *Xic* cells.

It will be assumed in this discussion that the PyCell Studio has been downloaded from Synopsys, and installed on your system, which also has *Xic* installed. The PyCell Studio works with Red Hat Enterprise Linux releases 5 and 6 (and equivalent). You must choose the same word size (32 or 64 bits) as your *Xic* installation. The installation location for PyCell Studio is selected by the user, and we will refer to this location as "`$CNI_ROOT`". For example, `$CNI_ROOT` might be `/usr/local/ciranova`.

Although your system will almost certainly have Python installed, it appears necessary to use the Python provided with the Studio. In Red Hat EL6, the Ciranova and stock Python version numbers are the same, but the libraries are apparently built with different options, and attempts at using the stock Python have failed (perhaps Synopsys will fix this?). You can, however, use your own OpenAccess installation if you have one and it is reasonably recent. You can probably also use OpenAccess from Cadence.

The first step is to make sure that the PyCell Studio installation is correct by following the steps in the `$CNI_ROOT/quickstart/README.txt` file.

Part of this procedure (step 3) is to source one of the startup files provided. This step sets the value of several environment variables, and forces the system to find the Ciranova Python instead of a local Python. It also installs the OpenAccess plug-in for Python. The user can customize this script if desired. It is necessary to source this file, or otherwise setup the environment as per the file, before starting *Xic*. After finishing, you will want to revert the environment to the previous state. Unfortunately, this is difficult. You may kill the window and start a new one.

A better way to run *Xic* in the Ciranova environment would be to write a script such as the following. Call it "`xic.cni`".

```
#! /bin/sh

CNI_ROOT=/usr/local/ciranova
source \$CNI_ROOT/quickstart/bashrc; xic -Tcni \$*
```

The `CNI_ROOT` line should be changed to the actual Ciranova installation location. After creating the file, make it executable with

```
chmod 755 xic.cni
```

Then, to run *Xic* in the Ciranova environment, just run this script instead. Since it runs in a sub-shell, the environment of the main shell is not corrupted. Any command line arguments are passed through.

Note that above *Xic* is started with a "`-Tcni`" option, which specifies to use the `xic_tech.cni` example technology file provided with *Xic*. This uses the `ReadDRF` and `ReadCniTech` directives to read display resource and technology files from the Ciranova installation. However, Ciranova provides a number of technology files, any you may want to try them. You will probably want to copy the `xic_tech.cni` file to your local directory, so that it can be edited easily.

Finally, you will need to set up your OpenAccess `lib.defs` to include the Ciranova libraries. The `lib.defs` file is a listing of the OpenAccess libraries available, very similar to the `cds.lib` file in Cadence. If no `lib.defs` file exists in the current directory, using a text editor create the file with a single line

`INCLUDE` *path/to/ciranova*`/quickstart/lib.defs`

The *path/to/ciranova* is the installation location, what we have called `$CNI_ROOT`. If there already is a `lib.defs` file, the line above should be added.

Once setup is complete, we can test it.

1. `prompt> ./xic.cni`
   Xic should start, and the "Using OpenAccess" and "Using Python" messages should appear in the console. The layer table will show perhaps unfamiliar layers, these have been obtained from the Ciranova technology file. There shouldn't be any error or warning message pop-ups.

2. Switch the editing context to a new, empty cell, if the current cell is not empty or is otherwise of value.

3. Click the **OpenAccess Libs** button in the **File Menu**, which will exist if OpenAccess is connected (the "Using OpenAccess" message appeared). This will bring up the **OpenAccess Libraries** panel. The following libraries will be listed.

```
IPL_cni130
cnVPcellLib
```

4. Click on the `IPL_cni130` line to select it, and press the **Contents** button. The **Listing** panel should appear, loaded up with names.
5. In the **Contents**, find the `Nmos2` entry, and click on it to select it.
6. Press the **Place** button in the **Contents** listing. The **Cell Placement Control** panel will appear. Press the **Place** button in this panel, and the **Parameters** pop-up will appear. There will be a double-line box "attached" to the mouse pointer.
7. Have a quick look at the **Parameters** panel. These are the pcell parameters that can be set. Feel free to enter some new values. The documentation for the `Nmos2` pcell will explain what the parameters are, though a few, such as `fingers`, `l`, and `w`, are obvious.
8. Click anywhere in the drawing window to place an instance. You should expand the view to show the instance content, press **Ctrl-x** for this. You can place more instances, perhaps with different parameters set. Press the **Esc** key when done.
9. Click on one of the instances to select it. Note that some of the sides of certain features are highlighted. These are stretch handles that can be dragged, to change the size of the feature. Try dragging a handle and note the effect.
10. Place a second instance of `Nmos2` so that it doesn't overlap ther first.
11. Move the second instance, place it so that one of the S/D contacts overlaps a contact of the first instance. Note that the overlapping contact has disappeared in both instances. This is auto-abutment. the two instances can be repositioned so as to exactly share the common edge, which implements a dual-gate transistor.
12. Press the **PCell Control** button in the **Edit Menu**, which will display the **PCell Control** pop-up. In the pop-up, change the **Auto-abutment mode** to **Mode 2 (with contact)**.
13. Move one of the cell instances well away from the other, note that both instances revert to the original form. Now drag and drop one of the instances over the other so that they share a contact, as before. This time, however, note that a common contact is retained.

This should be enough to get started, have fun!

5.7 Cadence^{TM} Compatibility

Limited compatibility with Cadence Virtuoso^{TM} is available on two levels. First, technology, display resource (DRF), and layer mapping files can be read directly by *Xic*. These files are generally provided in vendor-supplied process design kits intended for use with Cadence Virtuoso. Second, the OpenAccess plug-in allows *Xic* to access the Cadence libraries directly. Designs can be loaded into *Xic*, however presently they cannot be returned to Virtuoso without losing data required by Virtuoso.

For export to a Cadence environment, the **!dumpcds** command will create compatible technology and DRF files based on the *Xic* technology file in use.

Import of a Cadence technology environment is handled by three keywords which are given in the *Xic* technology file. In fact, a minimal technology file can consist of little more than these keywords. The keywords should appear in the order given, but otherwise can appear anywhere in the *Xic* technology file.

`ReadDRF` *drf_file*
: This reads the display resource file (DRF), which creates tables of layer colors, fill patterns, and similar for use in displays.

`ReadCdsTech` *techfile*
: This will read a Virtuoso ASCII technology file. The technology file contains the layer definitions, and usually quite a lot of technology information. From this, many of the *Xic* design rules and extraction keywords can be obtained.

`ReadOaTech` *library*
: This will obtain Virtuoso technology information directly from OpenAccess. The *library* is an OpenAccess library, listed in the `lib.defs` or `cds.lib` file. This obtains technology information by use of the OpenAccess plug-in. There should be no reason to use both this and `ReadCdsTech`, as they should retrieve the same information.

`ReadCdsLmap` *filename*
: The *filename* is the path to a Virtuoso layer mapping file. This provides GDSII layer/datatype numbers for the layers. This must appear in the *Xic* technology file after `ReadCdsTech`.

An *Xic* technology file can consist of these statements only. This will set the layers and their colors, fill patterns, and some or all of the electrical, extraction, and design rule information.

When a technology file is written with the **Save Tech** command, it will have the usual format and the lines described above are **not** included in the new file.

The ability to read the Lisp/Skill file format used by Virtuoso is provided by an internal Lisp parser. The parser is available to run general scripts through the **!lisp** command, though this has limited utility at present.

In the technology file, is is sometimes useful to enable debugging output from the Lisp parser. The following keyword enables this.

`LispLogging` [`y/n`]
: If this boolean keyword is set in the technology file, a log file will be generated when the Lisp parser is used. This can be used to track down issues when parsing Virtuoso-style input files. Asserting this keyword is equivalent to setting the Lisp logging in the **Logging Options** panel from the **Help Menu**, which otherwise can't be done before the technology file is read on program startup.

5.7.1 The Lisp Parser

The language supported here is similar to Lisp, and to the Cadence Skill language. The intention is not to replicate all features of these languages, but to provide a minimal subset of features for compatibility. The language will be referred to as "Lisp", but it should not be confused with the full-blown programming language.

The language differs from classic Lisp in that algebraic expressions within lists are evaluated, as in Skill. These reduce to a number token. One subtlety is detection of unary minus, for example `(2 -1)` could be interpreted as a list of two numbers, or one number (the difference). The parser will assume a unary minus if the preceding character is space or '`(`', and the following character is an integer or period followed by an integer.

One of the advantages of Lisp is the ease with which the syntax can be parsed. The basic data object is a "node", which has the form [*name*](*data* ...)

If a node has a *name*, there is no space between the name and the opening parenthesis. A named node is roughly equivalent to a function call. The *data* can be nodes, strings, or numerical expressions. The items are separated by white space. The *data* can use arbitrarily many lines in the input file.

Lisp variables are defined when assigned to, and have global scope unless declared in a `let` node, in which case their scope is within the `let` node, i.e., local.

A Lisp file consists of one or more named nodes. When the file is accessed with the **!lisp** command, each of the nodes is evaluated. The nodes must have names that are known to *Xic*. These are:

`main`
: The content of this node is evaluated. This is a special name for the "main" function of a script.

Built-in function name
: These are the basic Lisp functions and operator-equivalents.

Xic function name
: All of the *Xic* script functions will be recognized, however in Lisp the first character of these functions is always lower case. i.e., the `Edit` script function would be accessed as `edit`( ) in Lisp. Also, only *Xic* functions that take string or numeric arguments will work at present.

User-defined procedures
: These are Lisp functions defined by the user with the Lisp `procedure`( ) function.

Cadence compatibility name
: There is a growing number of node names that are used to interpret Cadence startup and control files (see 5.7).

A node name that can't be resolved will generate an error.

The parser uses the same numerical parser as the *WRspice* program, and hence recognizes numbers in the same (SPICE) format. All of the math functions based on the standard C library, as used in the native scripting language, are available.

The following built-in node names are recognized.

Operator Equivalents	
`expt`	`expt(x y)` $\Longleftrightarrow x\hat{\ }y$
`times`	`times(x y)` $\Longleftrightarrow x * y$
`quotient`	`quotient(x y)` $\Longleftrightarrow x/y$
`plus`	`plus(x y)` $\Longleftrightarrow x + y$
`difference`	`difference(x y)` $\Longleftrightarrow x - y$
`lessp`	`lessp(x y)` $\Longleftrightarrow x < y$
`leqp`	`leqp(x y)` $\Longleftrightarrow x <= y$
`greaterp`	`greaterp(x y)` $\Longleftrightarrow x > y$
`geqp`	`geqp(x y)` $\Longleftrightarrow x >= y$
`equal`	`equal(x y)` $\Longleftrightarrow x == y$
`nequal`	`nequal(x y)` $\Longleftrightarrow x \ != \ y$
`and`	`and(x y)` $\Longleftrightarrow x \ \&\& \ y$
`or`	`or(x y)` $\Longleftrightarrow x \ \mid\mid \ y$
`colon`	`colon(x y)` $\Longleftrightarrow' (xy) \Longleftrightarrow x : y$
`setq`	`setq(x y)` $\longleftrightarrow x = y$
Lists	
`'`	returns list of arguments
`list`	returns substituted list of arguments
`cons`	add element to front of list
`append`	append lists
`car`	return leading element of list
`cdr`	return list starting at second element
`nth`	return N'th element of list
`member`	return true if element in list
`length`	return length of list
`xCoord`	return first element of list
`yCoord`	return second element of list
Miscellaneous	
`main`	main function
`procedure`	define a procedure
`argc`	command line argument count
`argv`	command line argument list
`let`	variable scope container

5.7.2 The `ReadDRF` keyword

This technology file keyword is used to import a Cadence Virtuoso display resource (DRF) file into *Xic*. The syntax is

`ReadDRF` *drf_file*

The display resource file is generally provided by a process design kit intended to be used with Virtuoso. The file contains definitions of the layer colors and fill patterns, and other presentation attributes. Although the names may vary, the display resource file in one installation is named "`display.drf`"

The display resource file (DRF) ia a collection of "nodes", as understood by the Lisp parser. A named node has the form

name(*data* ...)

There can be no space between the node name and the opening parenthesis. The *data* are other Lisp nodes, strings, or numerical data or expressions. This can occupy arbitrarily many lines in the file. The DRF file consists of successive Lisp nodes, with names and expected content defined by Cadence.

The following top-level display resource Lisp nodes are understood by *Xic*. Presently, the only effect from these nodes is the creation of internal lists of data items, which are referenced by the nodes given in the Cadence ASCII technology file. Thus, reading in the display resource file has no effect on *Xic* operation other than providing display attributes for layers defined in the Cadence ASCII technology file.

`drDefineDisplay`
: This node is ignored.

`drDefineColor`
: For all entries with a display name of "`display`", the color is added to an internal color list. This internal list will be referenced in the technology file `techDisplays` node.

`drDefineStipple`
: For all entries with a display name of "`display`", the stipple pattern is added to an internal stipple list. This internal list will be referenced in the technology file `techDisplays` node.

`drDefineLineStyle`
: This node is ignored.

`drDefinePacket`
: For all entries with a display name of "`display`", the packet is added to an internal packet list. This internal list will be referenced in the technology file `techDisplays` node.

5.7.3 The `ReadCdsTech` keyword

This technology file keyword is used to import a Cadence Virtuoso ASCII technology file into *Xic*. The syntax is

`ReadCdsTech` *techfile*

The ASCII technology file is generally provided in process design kits intended for use with Virtuoso. The file name varies, but "`techfile`" and "`techfile.txt`" have been used. The file at minimum provides the list of layers used in the process. Generally, there is a wealth of technology information available, and the file can be quite large and complex.

If a display resource file is also being read, it should be read first. Other than this, `ReadCdsTech` can appear anywhere in the technology file, and will cause *Xic* to read information from the Cadence ASCII technology file given in *techfile*. This should be a full path to the file, unless the file is in the library search path.

The technology file is collections of "nodes", as understood by the Lisp parser. A named node has the form

name`(` *data* ... `)`

There can be no space between the node name and the opening parenthesis. The *data* are other Lisp nodes, strings, or numerical data or expressions. This can occupy arbitrarily many lines in the file. The file consists of successive Lisp nodes, with names and content that are defined by Cadence or OpenAccess. The nodes that are understood by *Xic* are described below.

Both Virtuoso 5.x and 6.x technology files can be read. Far more information can be obtained from 6.x (OpenAccess) technology files, however. This includes:

- Extraction technology keywords such as `Conductor`, `Via`, etc. (as are available from 5.x files) but additionally electrical/physical data such as `Thickness`, resistivity, and capacitance parameters are available.
- Design rules are generated from the "constraint groups".

This will provide a much more complete starting point from the technology file provided with a foundry kit. However, this still may be incomplete. For example, a typical technology file may provide thickness values for conductors only, not insulators.

Depending on the PDK, the imported design rules and derived layer definitions may require review and augmentation. The "real" design rules are likely provided in separate configuration files for Mentor Calibre, Cadence Assura, and/or others. In experience with one PDK, it was found that the rule set obtained through the OpenAccess technology database left a lot to be desired.

1. The very basic rules, such as `MinWidth` and `MinSpace` came through fine, including the spacing tables. Other simple rules also come through properly.
2. Derived layers come across fine, however within the syntax limitation, expresions are limited to a single operator, i.e., a form like "*layer operator layer*". Thus, a complex definition requires multiple derived layers for intermediate layers, which is acceptable. It was concerning, though, that the derived layers were not used anywhere within the technology file, such as in the constraints. There seemed also to be errors, for example one obvious place where "`'and`" was used where "`'or`" was clearly required.
3. The constraints helpfully included a design rule violation number, but were shown to be wrong when the rule was looked up. For example, One rule specified "(PP OR NP) Enclosure of PO ...", yet there were separate constraints "PP Enclosure PO..." and "NP Enclosure PO..." specified, which is wrong.
4. An attempt to DRC a known-clean layout with imported rules yielded a lot of bogus errors. Additional work would be necessary to obtain a "good" set of design rules.
5. As more tools use OpenAccess, perhaps there will be improvements in the rulesets provided through the OpenAccess technology database. At present, it appears that this is not primary to the serious DRC tools, but may be used by Virtuoso, possibly for editing feedback.

The tree below shows the hierarchy of the nodes that are recognized in the technology file. Most of these are ignored. Below we describe the nodes that are actually used, and what information they provide.

Below, nodes that were added for Virtuoso 6.1.4 are marked marked with an asterisk. The `constraintGroups` listing is greatly simplified, there is actually far more structure than indicated.

```
include
comment
controls
```

```
    techParams
    techPermissions
    viewTypeUnits *
    mfgGridResolution *
layerDefinitions
    techLayers
    techPurposes
    techLayerPurposePriorities
        techDisplays
        techLayerproperties
        techDerivedLayers *
layerRules
    functions *
    routingDirections *
    stampLabelLayers *
    currentDensityTables *
    viaLayers
    equivalentLayers
    streamLayers
viaDefs *
    standardViaDefs *
    customViaDefs *
constraintGroups *
    foundry *
        spacings *
            maxWidth
            minWidth
            minDiagonalWidth
            minSpacing
            minSameNetSpacing
            minDiagonalSpacing
            minArea
            minHoleArea
        viaStackLimits *
        spacingTables *
        orderedSpacings *
            minOverlap
            minEnclosure
            minExtension
            minOppExtension
        antennaModels *
        electrical *
    LEFDefaultRouteSpec *
        interconnect *
            maxRoutingDistance *
        routingGrids *
            verticalPitch *
            horizontalPitch *
            verticalOffset *
            horizontalOffset *
devices
```

```
        tcCreateCDSDeviceClass
        multipartPathTemplates *
        extractMOS *
        extractRES *
        symContactDevice
        ruleContactDevice
        symEnhancementDevice
        symDepletionDevice
        symPinDevice
        symRectPinDevice
        tcCreateDeviceClass
        tcDeclareDevice
    viaSpecs *
    physicalRules
        orderedSpacingRules
        spacingRules
        mfgGridResolution
    electricalRules
        characterizationRules
        orderedCharacterizationRules
    leRules
        leLswLayers
    lxRules
        lxExtractLayers
        lxNoOverlapLayers
        lxMPPTemplates
    compactorRules
        compactorLayers
        symWires
        symRules
    lasRules
        lasLayers
        lasDevices
        lasWires
        lasProperties
    prRules
        prRoutingLayers
        prViaTypes
        prStackVias
        prMastersliceLayers
        prViaRules
        prGenViaRules
        prTurnViaRules
        prNonDefaultRules
        prRoutingPitch
        prRoutingOffset
        prOverlapLayer
```

We mention below only the nodes from which information is extracted. Note that this is a mixture of 5.x and 6.x nodes, providing unified support for all current Virtuoso releases. In most cases, a node with an unrecognized name will produce a warning message. These can be ignored, the purpose is only to identify "new" information in the technology file that might be useful to parse.

`include`
This node contains a string, which is a path to another Lisp file. That file will be opened and read.

`controls/viewTypeUnits`
For `maskLayout`, if `microns`, the *Xic* database resolutions 1000, 2000, 5000, 10000, and 20000 are accepted.

`controls/mfgGridResolution`
This will set the *Xic* MfgGrid parameter.

`layerDefinitions/techLayers`
This associates OpenAccess layer numbers with layer names and abbreviations. These are recorded in the *Xic* layer database.

`layerDefinitions/techPurposes`
This associates OpenAccess purpose numbers with purpose names and abbreviations. These are recorded in the *Xic* layer database.

`layerDefinitions/techLayerPurposePriorities`
This contains a list of layer-purpose pairs, using layer and purpose names previously defined. Each layer-purpose pair is used to create an *Xic* layer. These are created in the order listed.

In Virtuoso, there is no distinction between physical and electrical layers as there is in *Xic*. All Virtuoso layers are taken as physical layers, except for the following internal Virtuoso layer numbers which with any purpose number will generate an *Xic* layer listed in both the electrical and physical layer tables in *Xic*.

Layer Number	Virtuoso Layer Name
228	`wire`
229	`pin`
230	`text`
231	`device`
236	`instance`
237	`annotate`

`layerDefinitions/techDisplays`
This will assign the colors and fill patterns to layers that exist in the *Xic* layer table. This references the internal packet, color, and stipple lists created from the display resource nodes. In addition, the initial visibility and selectability states are set here, as well as the Invalid flag.

`layerDefinitions/techLayerproperties`
This node provides some directly applicable parameters, which are read and added to the appropriate layer. These include `sheetResistance`, `areaCapacitance`, `edgeCapacitance`, and `thickness`. The thickness value is specified in angstroms, which is converted to microns. The capacitance value units are picofarads and microns, thus no conversion is required.

`layerDefinitions/techDerivedLayers`
The derived layers will be imported directly, with the expression converted to an *Xic* layer expression string. The expression given in this node type consists of a single operator and two layer names. The operator keywords which map to geometrical combinations (`'and`, `'or`, `'not`, and `'xor`) are accepted. Others are ignored.

`layerRules/routingDirections`
Layers found in this table are given the `Routing` attribute.

`layerRules/viaLayers`
The conducting layers are assigned the `Conductor` attribute. The via layer is assigned the `Via` attribute. This is in 5.x files only.

`layerRules/streamLayers`
A GDSII import/export mapping is applied for each layer given. This is in 5.x files only.

`viaDefs/standardViaDefs`
This identifies layers that are given the `Via` attribute. The metal layers that are referenced by the via are given the `Conductor` attribute. The standard via definition is imported, and will be available for via generation from the **Via Creation** panel from the **Edit Menu**.

`constraintGroups/foundry/spacings/maxWidth`
This identifies a `MaxWidth` rule.

`constraintGroups/foundry/spacings/minWidth`
This identifies a `MinWidth` rule.

`constraintGroups/foundry/spacings/minDiagonalWidth`
This will map to a `Diagonal` clause in a `MinWidth` rule.

`constraintGroups/foundry/spacings/minSpacing`
This maps to either a `MinSpace` rule (one layer given) or a `MinSpaceTo` rule if two layers are given.

`constraintGroups/foundry/spacings/minSameNetSpacing`
This provides the `SameNet` clause to a `MinSpace` or `MinSpaceTo` rule.

`constraintGroups/foundry/spacings/minDiagonalSpacing`
This provides the `Diagonal` clause to a `MinSpace` or `MinSpaceTo` rule.

`constraintGroups/foundry/spacings/minArea`
This identifies a `MinArea` rule.

`constraintGroups/foundry/spacings/minHoleArea`
This provides the dimension for area filtering in a `NoHoles` rule.

`constraintGroups/foundry/spacings/minHoleWidth`
This provides the dimension for minimum width filtering in a `NoHoles` rule.

`constraintGroups/foundry/spacingTables`
This provides tables of length, width, and spacing values, for size-dependent spacing rules. These tables are parsed and added to `MinSpace` and `MinSpaceTo` rules.

`constraintGroups/foundry/orderedSpacings/minEnclosure`
This maps to a `MinSpaceFrom` rule, with the source and target layers swapped. It provides the `Enclosed` clause, which applies when the target figure is completely surrounded by the source material. The alias `minEnclosureDistance` is also recognized.

`constraintGroups/foundry/orderedSpacings/minExtension`
This is almost identical with `minEnclosure`, but does not require that the target figure be fully surrounded. It maps to a `MinSpaceFrom` rule in the same manner, but sets the rule dimension, not the `Enclosed` value. The alias `minOverlapDistance` is also recognized.

`constraintGroups/foundry/orderedSpacings/minOppExtension`
This is handled similarly to the two rules above, but sets the `Opposite` clause of the `MinSpaceFrom` rule.

`constraintGroups/LEFDefaultRouteSpec/interconnect/maxRoutingDistance`
This provides the maxdist routing parameter (see A.6.4).

`constraintGroups/LEFDefaultRouteSpec/routingGrids/horizontalPitch`
`constraintGroups/LEFDefaultRouteSpec/routingGrids/verticalPitch`
These provide the pitch routing parameter (see A.6.4).

`constraintGroups/LEFDefaultRouteSpec/routingGrids/horizontalOffset`
`constraintGroups/LEFDefaultRouteSpec/routingGrids/verticalOffset`
These provide the `offset` routing parameter (see A.6.4).

`layerRules/routingDirections`
This provides the preferred routing direction.

`constraintGroups/foundry/spacings/minWidth`
This maps to the width routing parameter (see A.6.4).

5.7.4 The `ReadOaTech` keyword

This is similar to `ReadCdsTech`, however it retrieves the tech data from OpenAccess relative to a given library, instead of from the ASCII technology file. The syntax is

`ReadOaTech` *library*

The *library* must be listed in the OpenAccess library definitions file, named **`lib.defs`** or named **`cds.lib`** in Cadence installations. The OpenAccess plug-in is used to obtain the information, and of course must be available and set to connect to an OpenAccess database.

The technology information is extracted into a temporary Virtuoso ASCII technology file, which is then parsed by the equivalent of specifying `ReadCdsTech` with this file. The same file can be obtained from the print option of the **!oatech** command. This can be used to view the tech information that is being extracted.

5.7.5 The `ReadCdsLmap` keyword

This technology file keyword allows import of a Cadence Virtuoso layer mapping file. This file provides the layer/datatype numbers for the layers defined in the display resource file. It is important that these numbers be equivalent in *Xic* for success in transferring design data via GDSII or OASIS files. The file is generally provided within a process design kit. The name of the file will vary, in one case it is the name of the technology with a "**`.layermap`**" extension.

The syntax is

`ReadCdsLmap` *filename*

The *filename* is a path to the Virtuoso layer mapping file. This must appear in the *Xic* technology file after the `ReadCdsTech` line, as the layers must exist in the *Xic* database before they can be assigned a GDSII mapping.

5.7.6 Connecting to Cadence Installations

The OpenAccess plug-in (see 2.11) makes it possible for *Xic* to access Cadence cell libraries, by making use of the OpenAccess libraries provided with the Cadence installation.

When accessing Virtuoso design data, *Xic* should be provided with a consistent technology file. The Cadence compatibility features include the ability to read Virtuoso display resource, ASCII technology, and GDSII layer mapping files. These files are provided in the process design kit in use. The user should create a skeletal *Xic* technology file which will read these files. Then, layout appearance will be consistent between Virtuoso and *Xic*.

Compatibility and Setup

The present release of *Xic* is known to be compatible with Virtuoso 6.1.6 in terms of OpenAccess versioning. It is very likely compatible with earlier 6.1.x releases, but these have not been tested, though 6.1.4 has been verified with earlier *Xic* releases.

The installation location of the Cadence tools may be set in the environment variable CDSHOME. The user should verify that this variable is set in the environment. If not, the user must locate the installation directory for Cadence tools some other way.

Listing the installation directory, e.g.,

```
ls $CDSHOME
```

will provide a listing of files and subdirectories, which include the names "`tools`" and "`tools.lnx86`". In addition, there will be a subdirectory (perhaps more than one) with a name similar or identical to "`oa_v22.43.050`". This is OpenAccess. In this directory you will find a "`lib`" directory containing subdirectories with library files for 32 and 64-bit systems. The files of interest will match the *Xic* installation bit-width. The OpenAccess provided with Virtuoso 6.1.6 is newer than the publicly available version of OpenAccess that *Xic* is compiled against, but that does not appear to matter. If there are multiple OpenAccess versions present, probably the newest one (largest release numbers) should be used, but if problems are encountered other versions can be tried.

The full path to the directory containing the appropriate OpenAccess shared library files must be added to the system's library search string. On an example Cadence installation, the path, for 64-bits, is

```
$CDSHOME/oa_v22.43.050/lib/linux_rhel50_gcc44x_64/opt
```

In addition, callbacks may require that *Xic* have access to additional shared libraries supplied by Cadence. For 64-bits, this directory is

```
$CDSHOME/tools.lnx86/lib/64bit
```

Traditionally in Unix/Linux, the shared library search path is modified by setting the LD_LIBRARY_PATH environment variable. This variable provides additional locations for the system to search for needed shared libraries, in addition to system default locations that are implicit.

This variable can be used to set the search path, but in *Xic* there is a better way: set the XIC_LIBRARY_PATH environment variable instead. This is like LD_LIBRARY_PATH. but applies only to the *Xic* program. Setting LD_LIBRARY_PATH applies to all programs, whether they need the additional search locations or not.

The XIC_LIBRARY_PATH variable is most conveniently set in the user's shell startup file. The variable string consists of a list of directories, separated by colon (':') characters. The directories in the list are searched left-to-right to resolve shared library references, when a program is started. One should probably also include the value of the LD_LIBRARY_PATH in case that has been set for some other reason.

For our example, lines like the following should be added to the shell startup files. For `bash` and similar:

```
# Hook Xic to the Cadence OpenAccess library
XIC_LIBRARY_PATH=$CDSHOME/oa_v22.43.050/lib/linux_rhel50_gcc44x_64/opt
XIC_LIBRARY_PATH=$XIC_LIBRARY_PATH:$CDSHOME/tools/lib/64bit
export XIC_LIBRARY_PATH
```

and for C-shell:

```
# Hook Xic to the Cadence OpenAccess library
setenv XIC_LIBRARY_PATH $CDSHOME/oa_v22.43.050/lib/linux_rhel50_gcc44x_64/opt
setenv XIC_LIBRARY_PATH $XIC_LIBRARY_PATH:$CDSHOME/tools/lib/64bit
```

Similar commands can be given on the command line.

Once the new definitions apply, when *Xic* starts, the following message should appear on the console among the initial startup messages:

```
Using OpenAccess (oa.so).
```

If the message is not seen, try setting the XIC_PLUGIN_DBG environment variable and starting *Xic* again. Messages printed in the console window should indicate where the error occurs.

With OpenAccess successfully connected, the **File Menu** will contain the **OpenAccess Libs** button. If *Xic* was started in a directory with a `cds.lib` file, the libraries in the file should be listed in the pop-up. Probably, it is best when working with *Xic* to work from a different directory than when working with Virtuoso. If so, you will want to copy in your `cds.lib` file, which defines the Cadence libraries available. You can modify this copy with a text editor if desired. The libraries will be listed in the **OpenAccess Libraries** panel if they exist.

Express PCells

In Virtuoso, foundry devices are most likely represented as parameterized cells (pcells). These are cells with an internal script which generates a physical layout according to a set of device parameters.

Parameterized cells in the Cadence environment are most probably based on the Skill language and are not portable outside of a Cadence environment. However, Virtuoso provides a feature called "Express PCells" which caches pcell sub-masters in the user's home directory. A pcell sub-master is an ordinary cell, created from a pcell using a specific parameter set. The pcell cache provides the benefit that pcell evaluation is avoided, so that designs may be opened more quickly. A second advantage is that the cached sub-masters, unlike the pcells, can be exported.

Before a design containing Skill-based pcell instances can be fully loaded into *Xic*, the Express PCell feature must be enabled, and all of the pcell submasters must be cached.

One should be aware that if only a schematic is being imported into *Xic*, it isn't necessary to worry about pcells, as the pcell schematic symbol is available. Only the physical layout changes with different device parameters.

To enable Express PCells, the environment variable CDS_ENABLE_EXP_PCELL should be set to "`true`". Again, this is most conveniently done in the user's shell startup script. For `bash`:

```
export CDS_ENABLE_EXP_PCELL=true
```

For C-shell:

```
setenv CDS_ENABLE_EXP_PCELL true
```

From a Virtuoso Layout Editor window, the **Tools** menu will contain an **Express PCell Manager** button. This brings up a window allowing control of the feature. With the feature on, loading a design will populate the cache. It should then be possible to load the same design into *Xic*, with no unresolved pcell references. Note that when obtaining the pcell sub-masters through OpenAccess, a license checkout for the Cadence system occurs. Cadence will not export a sub-master from the cache without a license.

5.7.7 Importing a Design from Virtuoso

Once the OpenAccess database of a Cadence Virtuoso installation is connected to *Xic*, designs created in Virtuoso can be imported into *Xic*. Physical (layout) data should transfer without issues. Schematic and schematic symbol data will transform as electrical cells, some of which are devices. These will probably work successfully as *Xic* cells, but it is possible that a bit of intervention will be needed. It is disasterous if the cells are written back to Virtuoso. By default, Virtuoso libraries are read-only in *Xic* to prevent this from happening.

Xic obtains technology information from its own technology file, and (presently) not directly from OpenAccess. However, the same technology information should be available to *Xic* through direct reading of the appropriate `display.drf` and ASCII technology files. These files should be available in the process design kit in use.

The user's `cds.lib` file (or a copy) should exist in the current directory. This file will be used and updated by *Xic*. It is fine to share a `cds.lib` with an active Virtuoso installation, but it is probably better to maintain separate files, so that the *Xic* libraries, which are presently incompatible with Virtuoso, are invisible in Virtuoso.

If the OpenAccess database is connected, the **OpenAccess Libraries** panel, from the **OpenAccess Libs** button in the **File Menu**, will display the libraries that are defined in the `cds.lib` file. The button will appear in the **File Menu** only if an OpenAccess database is connected.

From the panel, one can select a library by clicking on the listing, and list the contents with the **Contents** button in the panel. Pressing the **Contents** button brings up a listing of the cells contained in the library.

Presently, *Xic* does not use "views" in the same manner as Virtuoso. Each of the listed cell names contain one or more of the following OpenAccess standard views, which are used to create the *Xic* cell. The `maskLayout` view contributes the physical data. The `schematic` view provides the electrical data, and the `schematicSymbol` view provides the *Xic* symbolic representation. Other views are ignored by *Xic*.

In the **Listing** panel, one can select cells bu clicking on a name. When a cell is selected, the **Open** button becomes active. Pressing this button will read that cell, and its hierarchy, into *Xic*. Note that it does not matter whether or not the library is “open” in the **OpenAccess Libraries** listing. The “open” status means that cells in the library will resolve instantiations as archive files are being read, but explicitly read cells, and subcells referenced in OpenAccess, are always read.

Before data can be successfully read into *Xic*, Virtuoso parameterized cells must be cached, using the Cadence Express PCells feature. *Xic* can not create super-masters for Virtuoso (Skill-based) pcells, but will import cached super-masters. The imported cell will be an ordinary cell in *Xic*, but will retain properties that identify the cell as originating as a Virtuoso pcell.

Once the hierarchy is read, it should appear visually very similar if not identical to the corresponding views in Virtuoso, if the appropriate technology has been accessed properly. Electrical cells will always have a symbolic representation, since in Virtuoso schematic instantiations are always symbolic, unlike in *Xic*.

If there are errors or warnings emitted during the import, the log file listing will appear. The user should inspect this and take appropriate action if needed.

With the design now local in *Xic*, it can be saved to disk in any of the formats supported by *Xic*. Initially, it is recommended saving the imported design as a collection of native cell files, into a clean directory. The **Export Control** panel from the **Convert Menu** can be used for this. Then, the devices can be “harvested”.

Initially, a number of the imported cells represent devices. These correspond to Virtuoso pcells, and have the same name. Except for some terminal devices that are created during translation as needed, the standard device library is not used. The imported devices serve the same purpose as the library devices, and will work in the same way. However, they will not appear in the device selection menus, and they are treated as ordinary cells in the hierarchy. By “harvesting” the devices, we will make “official” *Xic* devices out of them, allowing use in other designs, and remove them as ordinary cells in the imported hierarchy.

The following procedure can be used to identify the “device” cells. Bring up the **Cells Listing** panel from the **Cells Menu**. In the lower right-hand corner, select **Elec Cells** in the menu. Click the **Filter** button on the side of the listing, which will bring up the **Cell List Filter** panel. Make sure that the only box checked is the one next to **Device** (between **not** and **Device**). Then click **Apply**. The listing will now consist of the device cells only. You should save this list, using **Save Text** or otherwise.

After saving the imported design in a directory as native cell files, *Xic* can be exited. To harvest the devices, we will create a new directory (if needed), and move the device cells in our list from the directory containing our design to the new directory. We will then add the new directory as a reference in a local `device.lib` file, if this hasn’t been done previously. Then, next time we use *Xic*, the devices will be present in the device selection menus, and can be used in new schematics just as any other device. Specifically, suppose that you saved the design as native cell files in a directory named “`chip1`”, and you have another directory named “`devices`”. By hand, move each of the files in the list of devices from `chip1`<to `devices`. Then, add the `devices` directory to the `device.lib` file. The default system-wide `device.lib` is in the `startup` directory in the installation area (`/usr/local/xictools/xic/startup` by default). You can modify this file, or better copy this file to your current directory, and modify the copy. With a text editor, add a line to the end of the

`device.lib` file:

`Directory` */full/path/to/your/devices*

The second token should be the actual full path to the `devices` directory that you created. Note that in the future, all that you need to do to “install” a new device is to move the file into your `devices` directory.

Once finished, one can start *Xic* again, with the same technology file, and read in the top-level cell from the saved native cell files directory. The devices will be included, now resolved through the library mechanism. One may wish to save the design in an archive format such as OASIS or GDSII, which may be more convenient than the directory full of cell files. The archive file will not contain the devices. Be aware that to export the design to another *Xic* installation, the `devices` directory will have to be exported too.

5.8 Standard Vias

Xic provides a feature for creating and managing via objects used to connect between conducting layers in physical layouts. Although ordinary cells or cut-layer objects can be used for this purpose, use of standard vias offers some important advantages in many designs.

- The vias can contain the structure necessary so that proper use automatically satisfies design rule constraints.
- The vias are designed to allow a zero search depth for extraction, speeding this process.
- The creation of the "sub-master" cells that implement the vias is handled transparently by the system, removing the often large number of ordinary via cells from the cell listings. The via cells are no longer written in output, reducing file size and complexity.
- The vias are easily created from the **Via Creation** panel in the **Edit Menu** and can be placed immediately, which is quick and efficient.

In order for this feature to be available, one or more standard vias definitions must appear in the technology file. These will also be imported from a Cadence Virtuoso ASCII technology file if the `ReadCdsTech` keyword is used, and the Cadence database contains `standardViaDefs` definitions. The implementation of standard vias in *Xic* closely follows the implementation in OpenAccess, and tools such as Virtuoso that use OpenAccess.

The standard vias that are defined in the technology provide the default definitions for a via structure. Although commonly instantiated directly, more commonly variations are implemented. There are a number of parameters that define the via, and these can be changed by the user to produce a variant most suitable in the context where it will be used. For example, the cut can be arrayed when lower contact resistance is required.

The mechanism is similar to a parameterized cell (pcell). The standard vias defined in the technology can be considered as the super-masters. When a via of a certain configuration is requested, a "sub-master" cell for that configuration is created in memory, if it hasn't been created previously. The instances of the via will reference that sub-master. Like pcells, the masters are not written to disk. Instead, when a file containing via placements is read, the via sub-masters are created in memory as needed.

An exception is when shipping a layout to another system, such as to a mask vendor. The **Export Control** panel from the **Convert Menu** is used for this purpose. If the **Strip For Export** check box is checked or equivalently if the StripForExport variable is set, which should be true in this situation, the via (and pcell) sub-masters are included in the layout file. The foreign system will see these as ordinary cells. The **Include standard via cell sub-masters** check box or equivalently the ViaKeepSubMasters variable will likewise cause inclusion of the via sub-masters in output when set.

A standard via definition provides values for a number of parameters. Of these, the numerical values can be changed by the user to form a variant. The layers involved are immutable. Each standard via definition has a unique name assigned in the technology. This name can be any text which is suitable as a cell name. One convention is to use the layer names of the two conductors, top layer first, separated by an underscore, e.g., "`M2_M1`". The parameters and their effects are described with the **Via Creation** panel, from which the parameters can be set, and variants created and placed.

5.8.1 The Standard Via Property String

The `stdvia` property (number 7160) is applied to standard via instances and sub-masters. The OpenAccess translator will transparently convert these to the corresponding OpenAccess forms when writing to OpenAccess, and *vice-versa*. A string with very similar format to the property string is used by the `OpenViaSubMaster` script function. The property string syntax is described here.

There are actually two formats, that will be referred to as the old and new formats. The old format uses OpenAccess keywords and is friendly for humans, the new format uses a code and is more compact. *Xic* will always write the new format, but will read either format.

The property string consists of one or more space-separated text tokens. In either case, the first token is the name of the standard via, as given in the definition in the technology file. The remaining terms represent the numerical parameters that are different from the defaults given in the standard via definition. There need not be any additional tokens, in which case the via has all default values. More commonly, tokens follow the via name that provide alternate values.

In the old format, a token takes one of the forms

keyword`:`*value*
keyword`:`*value*`,`*value*

The *value* indicates an integer representing a dimension in internal units.

The new format assigns each numeric value a lower-case letter. A token consists of the letter, followed immediately by the numeric value in nanometers. The number is printed in a format which removes trailing zeros and decimal points.

new format key letter(s)	old format keyword
`a`	`CutWidth`
`b`	`CutHeight`
`c`	`CutRows`
`d`	`CutColumns`
`e,f`	`CutSpacing`
`g,h`	`Layer1Enc`
`i,j`	`Layer1Off`
`k,l`	`Layer2End`
`m,n`	`Layer2Off`
`o,p`	`OriginOff`
`q,r`	`Implant1Enc`
`s,t`	`Implant2Enc`

The new and old formats can **not** be mixed, all tokens must follow one format or the other. The cases with two letters correspond to the keywords with two values, and the values represent dimensions in the X and Y directions.

Examples:

```
M2_M1 CutRows:2 CutColumns:2 Layer1Enc:40,60
M2_M1 c2 d2 h60
```

The two strings are equivalent if 1) the database resolution is 1000 so that the internal unit is nanomenters, and 2) the default layer 1 enclosure in the X direction is 40nm.

When a sub-master is created, it is given a cell name that is the same as a new format property string with the space stripped out, and the characters '−' (minus) and '.' (period) replaced by '`m`' and '`p`', respectively.

The `OpenViaSubMaster` script function takes a string in almost the same format, the only difference is that the via name token is not present. Effectively, the via name is passed as the first argument, and the rest of the string (if anything) is passed as the second argument. Either new or old format is acceptable.

Chapter 6

The Help Menu: Obtain Program Documentation

The commands in the **Help Menu** provide documentation and help to *Xic* users.

The commands found in the **Help Menu** are summarized in the table below. The table provides the internal name for the command, and a brief description.

Help Menu			
Label	**Name**	**Pop-up**	**Function**
Help	`help`	**Help Viewer**	Show help, enter help mode
Multi-Window	`multw`	none	Set multi-window help mode
About	`about`	**About Panel**	Show version info
Release Notes	`notes`	**Text Editor**	Show release notes
Log Files	`logs`	**File Selection**	Provide access to log files
Logging	`dblog`	**Logging Options**	Set logging and debugging options

6.1 The Help Button: Obtain Help

Xic provides on-line context-sensitive help through activation of the **Help** button in the **Help Menu**. When this button is pressed, *Xic* enters help mode, and (unless suppressed) the help window appears with the default top-level topic. While help mode is active, information about commands and screen objects can be obtained by clicking with the left mouse button (button 1) on menu buttons or other screen objects. While in help mode, menu buttons will perform their normal functions rather than bringing up help text if the **Shift** key is held while the menu entry is activated. Help mode can be exited by pressing the **Esc** key while the pointer is in a drawing window, or by pressing the **Help** button a second time, but these will not remove the help window from the screen. Help mode is also exited when all help windows have been deleted, either with the **Quit** button in the help window **File** menu, or with window manager functions. If a help window is brought up with the keyboard **!help** command, *Xic* is not in help mode, thus menu buttons will have their normal functions.

If the variable HelpDefaultTopic is set (with the **!set** command or otherwise) to an empty string, pressing the **Help** button will not bring up the default top-level window. However, clicking on objects and buttons will bring up help topics as usual. One can also set this variable to a URL or database keyword, the content from which will appear in the initial window as the default topic.

Clicking on a colored HTML reference will bring up the text of the selected topic. If button 1 is used to click, the text will appear in the same window. If button 2 is used to click, a new help window containing the selected topic will appear.

The help system operates in one of two modes. The default mode is to use a single window for each new topic generated by pressing a command or menu button. In the multi-window implementation, which can be selected in *Xic* by selecting the **Multi-Window Mode** button in the **Help Menu**, or by setting the boolean variable HelpMultiWin with the **!set** command, a separate window is brought up for each press of a command button or menu item while in help mode. In either case, clicking on a link may or may not produce a new window, depending upon whether button 1 or button 2 was clicked.

Text shown in the viewer that is not part of an image can be selected by dragging with button 1, and can be pasted into other windows in the usual way.

The viewer can be used to display any text file or URL. In *Xic* and its derivatives, pressing the question mark key ("**?**") will prompt the user for text to display. The **!help** command has the same effect. In *WRspice*, the text to display can follow the "**help**" command keyword on the command line. The name given to the command, or to the **Open** command in the viewer's **File** menu, can be

- A keyword for an entry in the help database.
- A path to a file on the local machine. The file can be an image in any standard format, or HTML or plain text.
- An arbitrary URL accessible through the internet.

If the given name can be resolved, the resulting page will be displayed in the viewer. Also, the HTML viewer is sensitive as a drop receiver. If a file name or URL is dragged into the viewer and dropped, that file or URL is read into the viewer, after confirmation.

The ability to access general URLs should be convenient for accessing information from the internet while using *Xic* and *WRspice*. The prefix "`http://`" *must* be provided with the URL. Thus, for example,

```
help http://wrcad.com
```

will bring up the Whiteley Research web page in *Xic* or *WRspice*. The links can be followed by clicking in the usual way. Of course, the computer must have internet access for web pages to be accessible.

Be advised, however, that the "`mozy`" HTML viewer used in Unix/Linux releases is HTML-3.2 compliant with only a few HTML-4.0 features implemented, and has no JavaScript, Java, or Flash capabilities. A few years ago, this was sufficient for viewing most web sites, but this is no longer true. Most sites now rely on css styles, JavaScript, and other features not available in `mozy`. Most sites are still readable, to varying degrees, but without correct formatting.

The given URL is not relative to the current page, however if a '+' is given before the URL, it will be treated as relative. For example, if the viewer is currently displaying `http://www.foo.bar`, if one enters "`/dir/file.html`", the display will be updated to `/dir/file.html` on the local machine. If instead one enters "`+/dir/file.html`", the display will be loaded with `http://www.foo.bar/dir/file.html`.

The HTTP capability imposes some obvious limitations on the string tokens which can be used in the help database. These keywords should not use the '/' character, or begin with a protocol specifier such as "`http:`".

HTML files on a local machine can be loaded by giving the full path name to the file. Relative references will be found. HTML files will also be found if they are located in the help path, however relative references will be found only if the referenced file is also in the help path. If a directory is referenced rather than a file, a formatted list of the files in the directory is shown.

If a filename passed to the viewer has one of the following extensions, the text is shown verbatim. The (case insensitive) extensions for plain-text files are "`.txt`", "`.log`", "`.scr`", "`.sh`", "`.csh`", "`.c`", "`.cc`", "`.cpp`", "`.h`", "`.py`", "`.tcl`", and "`.tk`".

Holding **Shift** while clicking on an anchor that points to a URL which specifies a file on a remote system will download the file. References to files with extensions "`.rpm`", "`.gz`", and other common binary file suffixes will automatically cause downloading rather than viewing. When downloading, the **File Selection** panel will appear, pre-loaded with the file name (or "`http_return`" if the name is not known) in the current directory. One can change the saved name and the directory of the file to be downloaded. Pressing the **Download** button will start downloading. A pop-up will appear that monitors the transfer, which can be aborted with the **Cancel** button.

6.1.1 XicTools Update

The help system provides package management capability for the *XicTools* programs. Giving the keyword

```
:xt_pkgs
```

(note that the keyword starts with a colon) brings up a page listing the installed and available *XicTools* packages, for the current architecture. This requires internet access and http connectivity to `wrcad.com`.

One can select packages to download and optionally install by clicking on the check boxes. There are separate buttons to initiate downloading only, and downloading and installation. Package files, and the latest `wr_install` script if downloading, are downloaded to the current directory. Once installed, these files can be deleted.

The *XicTools* package management capability is available from the the internal help system in *Xic* and *WRspice*, and from the stand-alone *mozy* help browser.

6.1.2 The HTML Viewer

The help viewer windows provide access to the help system topics, and can display general HTML and image files.

There are three colored buttons in the menu bar of the viewer. The left-facing arrow button (back) will return to the previous topic shown in the window. The right-facing arrow button (forward) will advance to the next topic, if the back button has been used. The **Stop** button will stop HTTP transfers in progress.

There are four drop-down menus in the menu bar: **File**, which contains basic commands for loading and printing, **Options**, which contains commands for setting display attributes, **Bookmarks**, which allows saving frequently used references, and **Help** which provides documentation.

The **File** menu contains the following command buttons.

Open
The **Open** button in the **File** menu pops up a dialog into which a new keyword, URL, or file name can be entered.

Open File
The **Open File** button brings up the **File Selection** panel. The **Ok** button (green octagon) on the **File Selection** panel will load the selected file into the viewer (the file should be a viewable file). The file can also be dragged into the viewer from the **File Selection** panel.

Save
The **Save** button in the **File** menu allows the text of the current window to be saved in a file. This functionality is also provided by the **Print** button. The saved text is pure ASCII.

Print
The **Print** button brings up a pop-up which allows the user to send the help text to a printer, or to a file. The format of the text is set by the drop-down menu, with the current setting indicated on the menu button. The choices are PostScript in four fonts (Times, Helvetica, New Century Schoolbook, and Lucida Bright), HTML, or plain text. If the **To File** button is active, output goes to that file, otherwise the command string is executed to send output to a printer. If the characters "`%s`" appear in the command string, they are replaced with the temporary print file name, otherwise the temporary file name is appended to the string, separated by a space character.

Reload
The **Reload** button in the **File** menu will re-read the input file and redisplay the contents. This can be useful when writing new help text or HTML files, as it will show changes made to the input file. However, if you edit a "`.hlp`" file, the internally cached offsets for the topics below the editing point will be wrong, and will not display correctly. When developing a help text topic, placing it in a separate file will avoid this problem. One can also use the **!helpreset** command to update the file offset table. If the displayed object is a web page, the page will be redisplayed from the disk cache if it is enabled, rather than being downloaded again.

Old Charset
The help viewer uses the UTF-8 character set, which is the current standard international character set. However, older input sources may assume another character set, such as ISO-8859, that will display some characters incorrectly. If the user observes that some characters are missing or wrong in the display, setting this mode might help.

Make FIFO
This controls an obscure but unique feature. When the button is pressed, a named pipe, or FIFO, is created in the user's home directory. The name is "`mozyfifo`", or if this name is in use, an integer suffix is added to make the name unique. This is a special type of file, that has the property in this case that text written to this "file" will be parsed and displayed on the viewer screen.

The feature was developed for use in the stand-alone `mozy` program, for use as a HTML viewer for the `mutt` mail client. If an HTML MIME attachment is "saved" to the FIFO file, it will be displayed in the viewer.

The FIFO will be destroyed if this toggle button is pressed a second time, or when the help window exist normally. If the program crashes, the FIFO may be left behind and require manual removal.

Quit
The **Quit** button in the **File** menu removes the help window. This will exit help mode (where clicking on a command button brings up help) if there are no other help windows visible. Pressing the **Help** button in the **Help Menu** a second time or pressing the **Esc** key also exits help mode, though the help windows remain visible.

The **Options** menu presents a number of configuration and visual attribute choices to the user. These are described below.

Save Config

The **Save Config** button in the **Options** will save a configuration file in the user's home directory, named "`.mozyrc`". This file is read whenever a new help window appears, and sets various parameters, defaults, etc. This provides persistence of the options selected in the **Options** menu. Without an existing `.mozyrc` file, changes are discarded. If the file exists, it will be updated whenever a help window is dismissed.

Set Proxy

This button will create or manipulate a `.wrproxy` file in the user's home directory, which will provide a transport proxy url for internet access. The proxy will apply in all *XicTools* programs when connecting to the internet.

The `$HOME/.wrproxy` file contains a single line giving the internet url of the proxy server. The proxy server will be used to relay internet transactions such as checking for program updates, obtaining data or input files via http or ftp transport, and general internet access.

One can create a `.wrproxy` file by hand with a text editor. The general form is

`http://`*username*`:`*password*`@`*proxy.mydomain.com*`:`*port*

The format must be `http`, `https` is not supported at present. The *username* and *password* if needed are specified as shown, using the colon '`:`' and at-sign '`@`' as separators. The address can be a numeric ip quad, or a standard internet address. The port number is appended following a colon. No white space is allowed within the text.

When the menu button is pressed, a pop-up appears that solicits the proxy address. Here, the address is the complete token, as described above, but possibly without the port. The port number can be passed as a trailing number separated by white space, if it is not already given (separated by a colon). If no port number is given, the system will assume use of port number 80.

If the entry area is empty, any existing `.wrproxy` file will be moved to "`.wrproxy.bak`" in the user's home directory, effectively disabling use of a proxy. The behavior will be identical if the address consists of a hyphen '`-`'. An existing `.wrproxy.bak` file will be overwritten. If the hyphen is followed by some non-space characters, the `.wrproxy` file will be moved to a new file where the given characters serve as a suffix following a period. For example, if `-ZZ` is given, the new file would be "`.wrproxy.ZZ`" in the user's home directory. An existing file of that name will be overwritten.

If the argument consists of only a plus sign '`+`', if a file named "`.wrproxy.bak`" exists in the user's home directory, it will be moved to `.wrproxy`. An existing `.wrproxy` will be overwritten. If the '`+`' is followed by some non-space characters, the command will look for a file where the characters are used as a suffix, as above, and if found the file will be moved to `.wrproxy`.

Only the `.wrproxy` file will provide a proxy url, the other files are ignored. The renamed files provide convenient storage, for quickly switching between proxys, or no proxy.

Otherwise, if an address is given, the first argument must start with "`http:`" or an error will result.

Search Database

The **Search Database** button in the **Options** menu brings up a dialog which solicits a regular expression to use as a search key into the help database. The regular expression syntax follows POSIX 1003.2 extended format (roughly that used by the Unix `egrep` command). The search is case-insensitive. When the search is complete, a new display appears, with the database entries which contained a match listed in the "References" field. The library functions which implement the regular expression evaluation differ slightly between systems. Further information can be found in the Unix manual pages for "regex".

Find Text

The **Find Text** command enables searching for text in the window. A dialog window appears, into which a regular expression is entered. Text matching the regular expression, if any, is selected and scrolled into view, on pressing one of the blue up/down arrow buttons. The down arrow searches from the text shown at the top of the window to the end of the document, and will highlight the first match found, and bring it into view if necessary. The up button will search the text starting with that shown at the bottom of the window to the start of the document, in reverse order. Similarly, it will highlight and possibly scroll to the first match found. The buttons can be pressed repeatedly to visit all matches.

Default Colors

The **Default Colors** button in the **Options** menu brings up the **Default Colors** panel, from which the default colors used in the display may be set. The entries provide defaults which are used when the document being displayed does not provide alternative values (in a `<body>` tag). The defaults apply in general to help text.

The color entries can take a color name, as listed in the listing brought up with the **Colors** button, or a numerical RGB entry in any common format. The entries are the following:

Background color
Set the default background color used.

Background image
If set to a path to an image file in any standard image format, the image is used to tile the background.

Text color
The default color to use for text.

Link color
The default color to use for un-visited links.

Visited link color
The default color to use for visited links.

Activated link color
The default color to use for a link over which the user presses a mouse button.

Select color
The color to use as the background of selected text. This color can not be set from the document.

Imagemap border color
The color to use for the border drawn around imagemaps. This color can not be set from the document.

The **Colors** button brings up a panel which lists available named colors. Clicking on a name in this panel selects it, and enters the name into the system clipboard. The "paste" operation can then be used to enter the color name into an entry area. This may vary between systems, typically clicking on an entry area with the middle mouse button will paste text from the clipboard.

Pressing the **Apply** button will apply the new colors to the viewer window. Pressing **Dismiss** or otherwise retiring the panel without pressing **Apply** will discard changes. Changes made will **not** be persistent unless the **Save Config** button has been used to create a `.mozyrc` file, as mentioned above.

Set Font

The **Set Font** button in the **Options** menu will bring up a font selection pop-up. One can choose a typeface from among those listed in the left panel. The base size can be selected in the right panel. There are two separate font families used by the viewer: the normal, proportional-spaced font, and a fixed-pitch font for preformatted and "typewriter" text. Pressing **Apply** will set the currently selected font. The display will be redrawn using the new font.

In *Xic*, there are commands to set the font families:

`!helpfixed` [*family-size*]
`!helpfont` [*family-size*]

The format of the *family-size* argument depends upon the version of the GTK toolkit employed.

Cache group

A disk cache of downloaded pages and images is maintained. The cache is located in the user's home directory under a subdirectory named "`.wr_cache`". The cache files are named "`wr_cache`*N*"" where *N* is an integer. A file named "`directory`" in this directory contains a human-readable listing of the cache files and the original URLs. The listing consists of a line with internal data, followed by data for the cache files. Each such line has three columns. The first column indicates the file number *N*. The second column is 0 if the `wr_cache`*N* file exists and is complete, 1 otherwise. The third column is the source URL for the file. The number of files saved is limited, defaulting to 64. The cache only pertains to files obtained through HTTP transfer. This directory may also contain a file named "`cookies`" which contains a list of cookies received from web sites.

A page will not be downloaded if it exists in the cache, unless the modification time of the page is newer than the modification time of the cache file.

The **Don't Cache** button in the **Options** menu will disable caching of downloaded pages and images.

The **Clear Cache** button in the **Options** menu will clear the internal references to the cache. The files, however, are not cleared.

The **Reload Cache** button in the **Options** menu will clear and reload the internal cache references from the files that presently exist in the cache directory.

The **Show Cache** button in the **Options** menu brings up a listing of the URLs in the internal cache. Clicking on one of the URLs in the listing will load that page or image into the viewer. This is particularly useful on a system that is not continuously on-line. One can access the pages while on-line, then read them later, from cache, without being on-line.

No Cookies

Support is provided for Netscape-style cookies. Cookies are small fragments if information stored by the browser and transmitted to or received from the web site. The **No Cookies** button in the **Options** menu will disable sending and receiving cookies. With cookies, it is possible to view certain web sites that require registration (for example). It is also possible to view some commerce sites that require cookies. There is no encryption, so it is not a good idea to send sensitive information such as credit card numbers.

Images group

Image support is provided for gif, jpeg, png, tiff, xbm, and xpm. Animated gifs are supported as well. Images found on the local file system are always displayed immediately (unless debugging options are set in the startup file). The treatment of images that must be downloaded is set by this button group in the **Options** menu. One and only one of these choices is active. If **No Images** is chosen, images that aren't local will not be displayed at all. If **Sync Images** is chosen, images are downloaded as they are encountered. All downloading will be complete before the page is displayed. If **Delayed Images** is chosen, images are downloaded after the page is displayed. The display will be updated as the images are received. If **Progressive Images** is chosen, images are downloaded after the page is displayed, and images are displayed in sections as downloading progresses.

Anchor group
There are choices as to how anchors (the clickable references) are displayed. If the **Anchor Plain** button in the **Options** menu is selected, anchors will be displayed with standard blue text. If **Anchor Buttons** is selected, a button metaphor will be used to display the anchors. If **Anchor Underline** is selected, the anchor will consist of underlined blue text. The underlining style can be changed in the "`mozyrc`" startup file. One and only one of these three choices is active. In addition, if **Anchor Highlight** is selected, the anchors are highlighted when the pointer passes over them.

Bad HTML Warnings
If the **Bad HTML Warnings** button in the **Options** menu is active, messages about incorrect HTML format are emitted to standard output.

Freeze Animations
If the **Freeze Animations** button in the **Options** menu is active, active animations are frozen at the current frame. New animations will stop after the first frame is shown. This is for users who find animations distracting.

Log Transactions
If the **Log Transactions** button in the **Options** menu is active, the header text emitted and received during HTTP transactions is printed on the terminal screen. This is for debugging and hacking.

The **Bookmarks** menu contains entries to add and delete entries, plus a list of entries. The entries, previously added by the user, are help keywords, file names, or URLs that can be accessed by selecting the entry. Thus, frequently accessed pages can be saved for convenient access. Pressing the **Add** button will add the page currently displayed in the viewer to the list. The next time the **Bookmarks** menu is displayed, the topic should appear in the menu. To remove a topic, the **Delete** button is pressed. Then, the menu is brought up again, and the item to delete is selected. This will remove the item from the menu. Selecting any of the other items in the menu will display the item in the viewer. The bookmark entries are saved in a file named "`bookmarks`" which is located in the same directory containing the cache files.

6.1.3 The Help Database

The *XicTools* help system uses a fast hashed lookup table containing cached file offsets to the entry text. A modular database provides flexibility and portability. The files are located by default in the directories named "`help`" under the library tree, which is usually rooted at `/usr/local/xictools`. *Xic* and *WRspice* allow the user to specify the help search path through environment variables and/or startup files. All of the files with suffix "`.hlp`" in the directories along the help search path are parsed, and reference pointers added to the internal list, the first time the help command is issued in the application. In addition, other types of files, such as image files, which are referenced in the HTML help text may be present as well.

The help search path can be set in the environment with the variable XIC_HLP_PATH, and/or may be set in the technology file. The information on a given keyword can be accessed at any time using the "shell escape" command "**!help** *keyword*" in the prompt window.

The "`.hlp`" files have a simple format allowing users to create and modify them. Each help item is indexed by a keyword which should be unique in the database. The help text may be in HTML or plain text format. The file format is described in C.3.

6.1.4 Help System Forms Processing

There exists basic support for HTML forms. In *Xic*, HTML forms can be used as input sources for scripts. More information is available in 18.14.

When the form "Submit" button is pressed, a temporary file is created which contains the form output data. The file consists of key/value pairs in the following formats:

name=*single_token*
name="*any text*"

There is no white space around '=', and text containing white space is double-quoted. Each assignment is on a separate line.

The action string from the "`<form ...>`" tag determines how this file is used. The file is a temporary file, and is deleted immediately after use. If the action string is in the form "`action_local_`*xxxx*", then the form data are processed internally.

If the full path for the action string begins with "`http://`" or "`ftp://`", then the form data are encoded into a query string and sent to the location (though it is likely an error for ftp). Otherwise, the file will processed locally. This enables the output from the form to be processed by a local shell script or program, which can be very useful. The command given as the action string is given the file contents as standard input. The command standard output will appear in the HTML viewer window. Thus, one can create HTML form front-ends for favorite shell commands and programs.

6.1.5 Help System Initialization File

When a help window pops up, an initialization file is read, if it exists. This file is named "`.mozyrc`" and is sought in the user's home directory. This file is not created automatically, but is created or overwritten with the **Save Config** button in the **Options** menu of a help window. This need be done once only. It should be done if a `.mozyrc` file exists, but it is from a release branch earlier than 3.3. Once a `.mozyrc` file exists, it will be updated when leaving help, reflecting any setting changes.

Incidently "mozy" is the name of the stand-alone version of the HTML viewer/web browser available on the Whiteley Research web site.

6.2 The Multi-Window Button: Set Multi-Window Help Mode

When the **Multi-Window Mode** button in the **Help Menu** is set, in help mode, clicking on a menu item or screen object will pop up a new help window, rather than reusing a single existing window.

This menu item tracks the state of the HelpMultiWin variable.

6.3 The About Button: Program and Legal Info

The **About** button in the **Help Menu** brings up a text window which provides the *Xic* revision number and legal information. This window also appears when the key sequence **Ctrl-v** is entered, with the pointer in a drawing window.

6.4 The Release Notes Button: View Release Notes

The **Release Notes** button in the **Help Menu** brings up a text browser window loaded with the release notes for the current *Xic* release.

The release notes are installed by default in the directory `/usr/local/xictools/xic/docs`, and *Xic* searches this directory for the notes. *Xic* can be directed to look in a different directory in two ways. First, the environment variable XIC_DOCS_DIR can be set to the directory to search. Second, the variable DocsDir can be set (with the **!set** command) to the directory to search. The release notes describe bugs fixed and new features added to *Xic*, and should be read after a new release is installed. Also, they serve as supplements to the manual between printings. By policy, all updated information contained in the release note is incorporated into the help database for a given release.

6.5 The Log Files Button: Access Log Files

The **Log Files** button in the **Help Menu** brings up the **File Selection** panel pointing at the directory containing the log files. "Opening" one of the entries will bring up the **File Browser** loaded with the selected file.

The log files are kept in a temporary directory which is created when *Xic* is started. On normal exit, this directory is deleted, so if the user wishes to retain one or more of the log files, the files must be copied to a safe place. If *Xic* terminates unexpectedly, the directory is retained, and therefor the files are available for post-mortem debugging.

6.6 The Logging Button: Set Logging and Debugging Options

This **Logging** button in the **Help Menu** brings up the **Logging Options** panel, from which various logging and debugging options can be set. Probably, there is not much here that would be of interest to most users. Some users may find this useful for diagnosing problems, however.

The top half of the panel contains a number of check boxes, each with a description. Checking these boxes enables a debugging mode for the described subsystem or feature. This may involve additional consistency testing and messages. By default, these messages will go to the console window, unless a path to a file is entered into the **Message file** entry area, in which case messages will be saved in that file.

The bottom half of the panel enables logging output from the indicated subsystems, into the file whose name is given. These files will be created in the log files area, which is a temporary directory that is removed on normal program exit. The files in the log files area can be accessed with the **Log Files** button in the **Help Menu**.

This panel can also be brought up with the (undocumented) **!debug** command.

Chapter 7

The Side Menu: Geometry Creation

Xic has a "side" menu of buttons, typically displayed along the left edge of the application main window, next to the layer table. This menu contains buttons specific to editing, and is shown only when editing is enabled (meaning that it never appears in the *Xiv* feature set). The content of the menu differs between electrical and physical modes.

If the environment variable XIC_MENU_RIGHT is set when *Xic* starts, the menu and layer table will be placed along the right edge of the main application window. This might be more convenient for left-handed users. If the XIC_HORIZ_BUTTONS environment variable is set, the "side" menu buttons will instead be arrayed horizontally across the top of the main application window, above the top button menu.

This section describes in detail the commands available in the side menu in physical and electrical modes. These include commands for geometry creation and other frequently used operations.

Again, the side menu is only visible when cell editing is possible.

Side menu commands are executed by clicking with button 1 on the buttons. Typing the first few letters of the command name while pointing in a drawing window will also initiate a side menu command. The characters typed are displayed in the key press buffer area to the left of the prompt line in the main window, or in the upper-right corner of sub-window pop-ups. Commands can be exited by selecting the same or another command in most cases, or by pressing the **Esc** key.

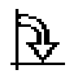
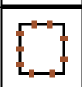
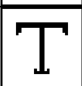
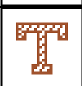
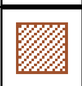
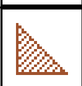
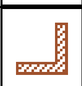
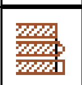
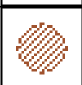
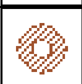
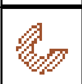
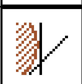
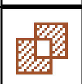
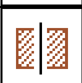
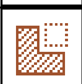
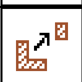
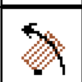
In the command descriptions, reference if often made to the "current transform". This is a rotation, reflection, and magnification specification for moved or copied objects, and for newly created subcells. The current transform is set with the pop-up produced by the **xform** button in the side.

Reference is also made to "selected" objects. Objects are selected by clicking the left mouse button (button 1) while pointing at the object, or by pressing and holding button 1 so that the object is enclosed in the rectangle formed with the press and release locations. Selecting a second time will deselect the objects, and all selected objects can be deselected with the **desel** button in the top button menu. Selected objects are displayed with a blinking highlighted border. Objects can also be selected with the **!select** command typed in the prompt area.

Reference is made to various commands that start with an exclamation point "**!**" such as "**!set**". These commands can be entered from the keyboard. Since most of these commands are used infrequently, they are not assigned command buttons. The most important of these commands is probably **!set**, since this allows certain variables to be set which control the behavior of some side menu commands. These "**!**" commands are described in chapter 19.

The tables below summarize the command buttons provided in the side menus in physical and electrical mode. Note that the side menu is different between physical and electrical modes, and that the operation of some commands which appear in both may differ slightly. These differences are noted in the descriptions. In the text, side menu commands are referenced by their internal names, since the command buttons contain an icon and not a label.

The side menu is not available in the *Xiv* feature set, and is invisible when certain modes are in effect, such as in CHD display mode, where editing is not allowed.

Physical Side Menu		
Icon	Name	Function
	`xform`	Set current transform
	`place`	Place subcells
	`label`	Create/edit labels
	`logo`	Create text object
	`box`	Create rectangles
	`polyg`	Create polygons
	`wire`	Create wires
	`style` menu	Set wire style
	`round`	Create disk objects
	`donut`	Create disk with hole
	`arc`	Create arcs
	`sides`	Set rounded granularity
	`xor`	Exclusive-OR objects
	`break`	Cut objects
	`erase`	Erase geometry
	`put`	Paste from yank buffer
	`spin`	Rotate objects

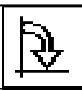
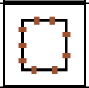
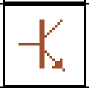
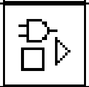
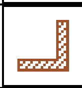
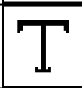
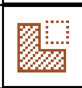
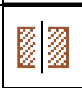
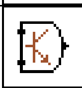
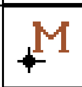
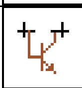
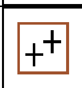
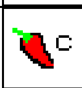
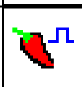
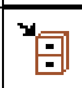
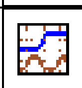
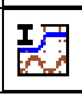
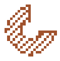
Electrical Side Menu		
Icon	Name	Function
	`xform`	Set current transform
	`place`	Place subcells
	`devs`	Show device menu
	`shapes` menu	Create outline object
	`wire`	Create wires
	`label`	Create/edit labels
	`erase`	Erase geometry
	`break`	Cut objects
	`symbl`	Set symbolic mode
	`nodmp`	Name wire nets
	`subct`	Set subcircuit contacts
	`terms`	Show terminals
	`spcmd`	Execute *WRspice* command
	`run`	Run *WRspice*
	`deck`	Save SPICE file
	`plot`	Plot SPICE results
	`iplot`	Set dynamic plotting

Table 7.1: Commands found in the side menu in physical and electrical modes.

7.1 The arc Button: Create Arcs



The **arc** command button allows the user to create arcs on the current layer. The **sides** button, or the **Sides** entry in the **shapes** menu in electrical mode, can be used to reset the number of segments used to represent the circle containing the arc. Press button 1 first to define the center. Subsequent presses, (or drag releases) define the inner and outer radii, the arc start angle, and the arc terminal angle. In physical mode, if the arc path width is set to zero, a round disk is created, as with the **round** button. If the angle given is 360 degrees, then the created figure is identical to that produced by the **donut** button. In electrical mode, the arc function is entered through the **arc** entry in the menu brought up with the **shapes** button. In this case, the arc path has no width, so that the inner and outer radii are equal and not separately definable. Arcs have no electrical significance, but can be used for illustrative purposes.

While the command is active in physical mode, the cursor will snap to horizontal or vertical edges of existing objects in the layout if the edge is on-grid, when within a small distance. When snapped, a small dotted highlight box is displayed. This makes it much easier to create abutting objects when the grid snap spacing is very fine compared with the display scaling. This feature can be controlled from the **Edge Snapping** group in the **Snapping** page of the **Grid Setup** panel.

In electrical mode, an arc is actually a wire, and as such should not be used on the SCED layer. If the current layer is the SCED layer, the arc will be created using the ETC2 layer, otherwise the arc will be created on the current layer. Although there is no error, arc vertices on the SCED layer are considered in the connectivity establishment, leading to inefficiency. If the user insists on the arc being on the SCED layer, the **Change Layer** command in the **Modify Menu** can be used to move it to that layer.

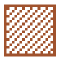
If the user presses and holds the **Shift** key after the center location is defined, and before the perimeter is defined by either lifting button 1 or pressing a second time, the current radius is held for x or y. The pointer location of the **Shift** press defines whether x is held (pointer closer to the center y) or y is held (pointer closer to the center x). This allows elliptical arcs to be generated. This similarly applies when defining the outer radii, so that the inner and outer surfaces can have different elliptical aspect ratios, though the outer radius must be larger than the inner radius at all angles.

The **Ctrl** key also provides useful constraints. Pressing and holding the **Ctrl** key when defining the radii produces a radius defined by the pointer position projected on to the x or y axis (whichever is closer) defined from the center. Otherwise, off-axis snap points are allowed, which may lead to an unexpected radius on a fine grid. When defining the angles of arcs with the **Ctrl** key pressed, the angle is constrained to multiples of 45 degrees. Ordinarily, the arc angle snaps to the nearest snap point.

When the command is expecting a mouse button press to define a radius, the value as defined by the mouse pointer (in microns) is printed in the lower left corner of the drawing window, or the X and Y values are printed if different. Pressing **Enter** will cause prompting for the value(s), in microns. If one number is given, a circular radius is accepted, however one can enter two numbers separated by space to set the X and Y radii separately.

Similarly, the angles are displayed, and can be entered in this manner. Prompts can be obtained for the start and end angles separately. The angle should be entered in degrees. Zero degress points along the X axis, and positive angles advance clockwise.

7.2 The box Button: Create Rectangles



The **box** command button allows creation of boxes (rectangles) on the currently selected layer. The box can be defined by either clicking button 1 on two diagonal corners, or by pressing button 1 to define the first corner, dragging, then releasing button 1 to define the second corner. The outline of the box is ghost-drawn during creation. The new box will be merged with or clipped to existing boxes on the same layer, unless this feature has been suppressed.

While the command is active in physical mode, the cursor will snap to horizontal or vertical edges of existing objects in the layout if the edge is on-grid, when within two pixels. When snapped, a small dotted highlight box is displayed. This makes it much easier to create abutting objects when the grid snap spacing is very fine compared with the display scaling. This feature can be controlled from the **Edge Snapping** group in the **Snapping** page of the **Grid Setup** panel.

In physical mode, boxes can also be created from the **Show/Select Devices** panel from the **Device Selections** button in the **Extract Menu**. The **Enable Measure Box** button provides a means of creating boxes of a specific size to match electrical requirements, for example to create rectangular resistor bodies for a given resistance. Boxes can be created whether or not the electrical layer parameters are used or present.

In physical mode while the **box** command is active, holding down the **Ctrl** key while clicking on a subcell will paint the area of the subcell with the current layer.

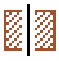
In electrical mode, the box command is available by selecting the **box** function in the **shapes** menu. If the current layer is the SCED layer, the box will be created using the ETC2 layer, otherwise the box will be created on the current layer. It is best to avoid use of the SCED layer for other than active wires, for efficiency reasons, though it is not an error. The **Change Layer** command in the **Modify Menu** can be used to change the layer of existing objects to the SCED layer, if necessary. The outline style and fill will be those of the rendering layer. Boxes have no electrical significance, but can be used for illustrative purposes.

The **box**, **erase**, and **xor** commands participate in a protocol that is handy on occasion.

Suppose that you want to erase an area, and you have zoomed in and clicked to define the anchor, then zoomed out or panned and clicked to finish the operation. Oops, the **box** command was active, not **erase**. One can press **Tab** to undo the unwanted new box, then press the **erase** button, and the **erase** command will have the same anchor point and will be showing the ghost box, so clicking once will finish the erase operation.

The anchor point is remembered, when switching directly between these three commands, and the command being exited is in the state where the anchor point is defined, and the ghost box is being displayed. One needs to press the command button in the side menu to switch commands. If **Esc** is pressed, or a non-participating command is entered, the anchor point will be lost.

7.3 The break Button: Cut Objects



The **break** button is used to divide objects along a horizontal or vertical line. The command operates on boxes, polygons, and wires. If one or more of those objects was previously selected, the break command will operate on those selections. Otherwise, the user is asked to select objects to break. The user is then asked to click to divide the selected objects along the break line, which is attached to the pointer and ghost-drawn. The orientation of the break line is either horizontal or vertical, which can be toggled by pressing either the / (forward slash) or \ (backslash) keys when the break line is visible. The **break** command is useful when one wants to relocate or create a subcell from pieces of an existing design.

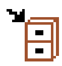
While the command is active in physical mode, the cursor will snap to horizontal or vertical edges of existing objects in the layout if the edge is on-grid, when within two pixels. When snapped, a small dotted highlight box is displayed. This makes it much easier to create abutting objects when the grid snap spacing is very fine compared with the display scaling. This feature can be controlled from the **Edge Snapping** group in the **Snapping** page of the **Grid Setup** panel.

When the **break** command is at the state where objects are selected, and the next button press would initiate the break operation, if either of the **Backspace** or **Delete** keys is pressed, the command will revert the state back to selecting objects. Then, other objects can be selected or selected objects deselected, and the command is ready to go again. This can be repeated, to build up the set of selections needed.

At any time, pressing the **Deselect** button to the left of the coordinate readout will revert the command state to the level where objects may be selected to break.

The undo and redo operations (the **Tab** and **Shift-Tab** keypreses and **Undo**/**Redo** in the **Modify Menu**) will cycle the command state forward and backward when the command is active. Thus, the last command operation, such as initiating the break by clicking, can be undone and restarted, or redone if necessary. If all command operations are undone, additional undo operations will undo previous commands, as when the undo operation is performed outside of a command. The redo operation will reverse the effect, however when any new modifying operation is started, the redo list is cleared. Thus, for example, if one undoes a box creation, then starts a break operation, the “redo” capability of the box creation will be lost.

7.4 The deck Button: Save SPICE File



The **deck** command, available only in electrical mode, creates a SPICE file of the current circuit hierarchy. The file name is prompted for, as is an analysis string. If an analysis string is given, it will be included in the SPICE file after prepending a ‘.’, unless it happens to start with “run”, in which case it is ignored. If a plot string has been created with the **plot** command, it will also be included as a `.plot` line.

Unless the variable SpiceListAll is set (with the **!set** command), only devices and subcircuits that are “connected” will be included in the SPICE file. A device or subcircuit is connected if any of the following is true:

- The subcircuit has a global node.
- The device or subcircuit has two or more non-ground connections.
- The device or subcircuit has one non-ground connection and one or more grounds.

- The device or subcircuit has one non-ground connection and no opens.
- The subcircuit has a non-ground connection.

Note that it is possible for a subcircuit to have no connections on the `.subckt` line, if it contains a global node. For example, the subcircuit might consist of a decoupling capacitor to ground, from a global power supply node (e.g., "`vdd!`").

Node names will be assigned according to the node name mapping (see 7.11 currently in force.

After the new file is created, the user is given the option of viewing it in a **File Browser** window.

If the variable CheckSolitary is set (with the **!set** command) then a warning will be issued if nodes are found with only one connection.

7.5 The devs Button: Device Menu

The **devs** button appears only in electrical mode, and pressing this button will toggle the display of the device selection menu.

There are three styles of the device menu. The default style contains a menu bar with four entries: **Devices**, **Sources**, **Macros**, and **Terminals**. Each brings up a sub-menu containing names of library "devices", that fall into each category.

The second menu style is similar, but the menu bar contains the first letter of the device name (not the SPICE key).

In either of these styles, pressing and holding button 1 while the pointer is over one of the menu bar buttons will pop up a menu of device names. Moving the pointer down the menu will highlight the entry under the pointer. A selection can be made by releasing the button.

The third style is the pictorial menu, which displays the schematic symbol of each available device, in alphabetical order. Clicking on one of the device images will establish the selection.

Each menu style contains a button from which the style can be cycled.

After a selection is made, the device symbol will be ghost-drawn and attached to the pointer, and the device will be placed at positions where the user clicks in the drawing windows. The device is positioned such that the reference terminal is located at the point where the user clicked. Devices are placed according to the current transform, which is defined from the pop-up produced by the **xform** button in the side menu.

When the menu becomes active, the current transform is cleared. The current transform is saved in Register 0 and cleared when switching between devices to place or upon pressing the **Desel** button. Pressing the **Forward Slash** button swaps the current and saved transforms.

The devices available and other details depend upon the definitions in the device library file. By default, this file is named "`device.lib`", and is located in the installation startup directory, but this can be superseded by a custom file of the same name which is found in the library search path ahead of the default file.

The present device menu style tracks, and is tracked by, the DevMenuStyle variable. This variable can be set (with the **!set** command) to an integer 0–2. If 0 or unset, the categorized layout is used. If

1, the alphabetized variation is used, and 2 specifies the pictorial menu. This variable tracks the style of the menu, and resets the style when set.

The following table lists the devices found in the device library file supplied with *Xic*.

Name	**Description**
	Contact Devices
gnd	Ground Contact
gnde	Alternative Ground Contact
tbar	Contact Terminal
tblk	Alternative Contact Terminal
tbus	Bus Contact Terminal
	SPICE Devices
res	Resistor
cap	Capacitor
ind	Inductor
mut	Mutual Inductor
isrc	Current Source
vsrc	Voltage Source
dio	Junction Diode
jj	Josephson Junction
npn	NPN Bipolar Transistor
pnp	PNP Bipolar Transistor
njf	N-Channel Junction FET
pjf	P-Channel Junction FET
nmos1	N-Channel MOSFET, 4 Nodes
pmos1	P-Channel MOSFET, 4 Nodes
nmos	N-Channel MOSFET, 3 Nodes
pmos	P-Channel MOSFET, 3 Nodes
nmes	N-Channel MESFET
pmes	P-Channel MESFET
tra	Transmission Line
ltra	Transmission Line (LTRA Compatible)
urc	Uniform RC Line
vccs	Voltage-Controlled Current Source
vcvs	Voltage-Controlled Voltage Source
cccs	Current-Controlled Current Source
ccvs	Current-Controlled Voltage Source
sw	Voltage-Controlled Switch
csw	Current-Controlled Switch
	Misc.
opamp	Example Macro
vp	Current Meter

The colors used in the pictorial device menu can be changed by setting the Special GUI Colors (see A.8.3) listed below. This can be done in the technology file, or with the **!setcolor** command.

variable	purpose	default
GUIcolorDvBg	background	gray90
GUIcolorDvFg	foreground	black
GUIcolorDvHl	highlight	blue
GUIcolorDvSl	selection	gray80

7.5.1 Terminal Devices

The following are not "real" devices, though they appear in the device menu and can be placed in a drawing. Their purpose is to establish connectivity.

Ground Device

The gnd device is used to connect to node 0, which is always taken as the reference (ground) node in SPICE. This can be placed in the main circuit and subcircuits.

The device library may contain multiple, functionally identical "ground" devices, that differ only visually. In the library, any device that has no name property and exactly one node property is taken as a ground device.

Alternative Ground Device

The gnde device is used to connect to node 0, which is always taken as the reference (ground) node in SPICE. This can be placed in the main circuit and subcircuits. This is functionally identical to the gnd device, but differs visually.

Terminal Device

The tbar, tblk, ttri, and txbox are "terminal devices" from the default device library. These devices behave identically, and differ only in appearance. Each device has an associated label (with text defaulting to the device name) which can be changed by the user by selecting the label and pressing the **label** button in the side menu. The label will supply a name, which will be applied to a connected net. All nets connected to a terminal device with the same name are taken as being connected together.

This will not tie nets between the main circuit and subcircuits, or between subcircuits, unless the terminal name is also a global net name. If not global, the scope is within the cell only. See 7.11 for more information about net name assignments.

Internally, the device will reconfigure itself as a scalar or multi-contact device according to the label. Older *Xic* releases provided a tbus terminal, which is no longer compatible.

The name applied to a net via a terminal device is handled identically to a name obtained from a wire label.

Bus Terminal Device

The tbus terminal device was provided as a bus terminal in older *Xic* releases. It is no longer compatible or supported, and must be replaced by a current terminal device in legacy schematics.

7.5.2 SPICE Devices

These devices correspond to element lines in SPICE output. In general, they reflect the generic SPICE syntax.

Resistor Device

The res device is a two-terminal resistor. Typically, a value property is added to specify resistance. Alternatively, a model property can be added to specify a resistor model. If a model property is assigned, then a param property can be used to supply the geometric or other parameters.

The '+' symbol in the representation accesses a branch property that returns a hypertext expression consisting of the voltage across the resistor divided by the resistance in ohms, yielding the current through the resistor. The 'O' that follows the resistance is the 'ohms' unit specifier, and *not* an extra zero.

Capacitor Device

The cap device is a two-terminal capacitor. Typically, a value property is added to specify capacitance. Alternatively, a model property can be added to specify a capacitor model. If a model property is assigned, then a param property can be used to supply the geometric parameters. In either case, the param property can be used to provide initial conditions.

The '+' symbol in the representation accesses a branch property that returns a hypertext expression consisting of the capacitance value times the time-derivative of the voltage across the capacitor, yielding the capacitor current.

Inductor Device

The ind device is a two-terminal inductor. A value property should be added to specify inductance. A param property can be used to provide initial conditions.

The '+' symbol in the representation accesses a branch property that returns a hypertext link to the inductor current vector.

Mutual Inductor

The mut device provides support for mutual inductors. The mut device is never placed. When the mut device is selected in the device menu, rather than selecting a device for placement as do the other selections, a command mode is entered which allows existing inductors to be selected into mutual inductor pairs.

When the mut device is selected, an existing pair of coupled inductors (if any have been defined) is shown highlighted, and the SPICE coupling factor printed. The arrow keys cycle through the internal list of coupled inductor pairs, or a pair may be selected by clicking on one of the inductors or the coefficient label with button 1. At any time, pressing the '**a**' key will allow addition of a mutual inductor pair. The same effect is obtained by clicking on a non-mutual inductor with button 1. The user is asked to click on the two coupled inductors (if '**a**' entered or there are no existing mutual inductors), or the second inductor (if the user clicked on an inductor), and then to enter the coupling factor. The coupling factor can be any string, so as to allow shell variable expansion in *WRspice*, but if it parses as a number it must be in the range between -1 and 1.

Pressing the '**d**' key will delete the mutual inductance specification for the two inductors currently shown.

Pressing the '**k**' key will prompt for a new value of the coupling factor for the mutual inductors shown, as will clicking on the coefficient label in a drawing window. When entering the coefficient string, one can enter either the form *name*=*coef_string*, or simply the coefficient string. In the first case, the *name* will provide an alternate fixed name for the mutual inductor in SPICE output. This can be any alphanumeric name, but should start with 'k' or 'K' for SPICE. If no name is given, *Xic* will assign a name consisting of `K` followed by a unique index integer.

One can also change the coefficient string and/or name with the **label** button in the side menu. Again, the label text can have either of the forms described above.

Pressing the **Esc** key terminates this (and every) command. One can back out of the operation if necessary with **Tab** (undo), as usual.

Current Source

The isrc device is a general current source. A value and/or param property can be added to specify the value, function, or other parameters required by the source.

The arrow head in the representation accesses a branch property that returns a hypertext link to the current in the form "`@`*name*`[c]`". A `.save` line for this vector is automatically added to the SPICE output.

Voltage Source

The vsrc device is a general voltage source. A value and/or param property can be added to specify the value, function, or other parameters required by the source.

The '+' symbol in the representation accesses a branch property that returns a hypertext link to the current vector when clicked on.

Current Meter

In SPICE, voltage sources are often used as "current meters", as the current through a voltage source is saved with the simulation result vectors, and can be plotted or printed. The vp device is actually a voltage source (identical to a `vsrc` device) however the symbol size is tiny, so that it can be more easily added to an existing schematic for use as a current meter. The symbol contains a hot spot in the representation that accesses a branch property that returns a hypertext link to the current vector when clicked on.

Junction Diode

The dio device is a junction diode. A model property should be added to specify a diode model. A param property can be added to specify additional parameters.

The diode contains no hidden targets.

Josephson Junction

The jj device is a Josephson junction. A model property should be added to specify a Josephson junction model. A param property can be added to specify additional parameters.

The '+' symbol in the representation accesses the phase node of the Josephson junction. The "voltage" on this node is equal to the junction phase, in radians.

NPN Bipolar Transistor

The npn device is an npn bipolar transistor. A model property should be added to specify a bipolar transistor model. A param property can be added to specify additional parameters.

The bipolar transistor contains no hidden targets.

PNP Bipolar Transistor

The pnp device is a pnp bipolar transistor. A model property should be added to specify a bipolar transistor model. A param property can be added to specify additional parameters.

The bipolar transistor contains no hidden targets.

N-Channel Junction FET

The njf device is an n-channel junction field-effect transistor. A model property should be added to specify a JFET model. A param property can be added to specify additional parameters.

The JFET contains no hidden targets.

P-Channel Junction FET

The pjf device is a p-channel junction field-effect transistor. A model property should be added to specify a JFET model. A param property can be added to specify additional parameters.

The JFET contains no hidden targets.

N-Channel MOSFET, 4 Nodes

The nmos1 device is a 4-terminal n-channel MOSFET (drain, gate, source, bulk). A model property should be added to specify a MOS model, suitable for 4-terminal devices. Some of the MOS models provided in *WRspice*, for SOI devices, use more than four terminals and will not work with this device. It is left as an exercise for the user to create a modified device suitable for use with these models. A param property can be added to specify additional parameters.

This device contains no hidden targets.

P-Channel MOSFET, 4 Nodes

The pmos1 device is a 4-terminal p-channel MOSFET (drain, gate, source, bulk). A model property should be added to specify a MOS model, suitable for 4-terminal devices. Some of the MOS models provided in *WRspice*, for SOI devices, use more than four terminals and will not work with this device. It is left as an exercise for the user to create a modified device suitable for use with these models. A param property can be added to specify additional parameters.

This device contains no hidden targets.

N-Channel MOSFET, 3 Nodes

The nmos device is an n-channel MOSFET variation that contains three visible nodes (drain, gate, source). The bulk node is connected to an internal global node named "NSUB". To use this device, the circuit should contain a voltage source tied to a terminal device with label "NSUB" to provide substrate bias to all devices of this type. This simplifies the schematic by hiding the substrate connection to each transistor.

A model property should be added to specify a MOS model, suitable for 4-terminal devices. Some of the MOS models provided in *WRspice*, for SOI devices, use more than four terminals and will not work with this device. It is left as an exercise for the user to create a modified device suitable for use with these models. A param property can be added to specify additional parameters.

This device contains no hidden targets.

P-Channel MOSFET, 3 Nodes

The pmos device is a p-channel MOSFET variation that contains three visible nodes (drain, gate, source). The bulk node is connected to an internal global node named "PSUB". To use this device, the circuit should contain a voltage source tied to a terminal device with label "PSUB" to provide substrate bias to all devices of this type. This simplifies the schematic by hiding the substrate connection to each transistor.

A model property should be added to specify a MOS model, suitable for 4-terminal devices. Some of the MOS models provided in *WRspice*, for SOI devices, use more than four terminals and will not work with this device. It is left as an exercise for the user to create a modified device suitable for use with these models. A param property can be added to specify additional parameters.

This device contains no hidden targets.

N-Channel MESFET

The nmes device is an n-channel MESFET. A model property should be added to specify a MESFET model. A param property can be added to specify additional parameters.

The MESFET contains no hidden targets.

P-Channel MESFET

The pmes device is a p-channel MESFET. A model property should be added to specify a MESFET model. A param property can be added to specify additional parameters.

The MESFET contains no hidden targets.

Transmission Line

The tra device is a general transmission line. In *WRspice*, this can be lossy or lossless, and may access a model. In other versions of SPICE, this is a lossless line with no model. A model property can be added to specify a transmission line model. A param property can be added to specify additional parameters.

The transmission line contains no hidden targets.

Transmission Line (LTRA compatibility)

The ltra device is a general transmission line. In *WRspice*, this can be lossy or lossless, and is basically the same as the tra device, but defaults to a convolution approach if lossy. In other versions of SPICE, this is a lossy line that requires a model. A model property can be added to specify a transmission line model. A param property can be added to specify additional parameters.

The transmission line contains no hidden targets.

Uniform RC Line

The urc device is a lumped-approximation RC line. A model property should be added to specify a urc model. A param property can be added to specify additional parameters.

The urc line contains no hidden targets.

Voltage-Controlled Current Source

The vccs device is a voltage-controlled dependent current source. A value and/or param property can be added to specify the gain, or other parameters required by the dependent source. Since all four nodes are specified, the two-node variants supported by *WRspice* are not supported by this device.

The VCCS contains no hidden targets.

Voltage-Controlled Voltage Source

The vcvs device is a voltage-controlled dependent voltage source. A value and/or param property can be added to specify the gain, or other parameters required by the dependent source. Since all four nodes are specified, the two-node variants supported by *WRspice* are not supported by this device.

The VCVS contains no hidden targets.

Current-Controlled Current Source

The cccs device is a current-controlled dependent current source. A devref property can be used to specify the name of the controlling voltage source or inductor in the common case. A value and/or param property should be added to specify gain, or other parameters required by the dependent source. This device supports all of the variants supported in *WRspice*.

The CCCS contains no hidden targets.

Current-Controlled Voltage Source

The ccvs is a current-controlled dependent voltage source. A devref property can be used to specify the name of the controlling voltage source or inductor in the common case. A value and/or param property should be added to specify the gain, or other parameters required by the dependent source. This device supports all of the variants supported in *WRspice*.

The CCVS contains no hidden targets.

Voltage-Controlled Switch

The sw device is a voltage-controlled switch. A model property should be added to specify a switch model. A param property can be added to specify additional parameters.

This device contains no hidden targets.

Current-Controlled Switch

The csw device is a current-controlled switch. A devref property must be used to specify the name of the controlling voltage source or inductor. A model property should be added to specify the switch model. A param property can be added to specify additional parameters.

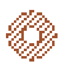
This device contains no hidden targets.

Example Opamp Macro

The opamp device is an example "black box" device that expands into a subcircuit. It has a predefined model parameter which gives the subcircuit name (which is resolved in the model library). No properties are required.

This device contains no hidden targets.

7.6 The donut Button: Create Donut Object



The **donut** button appears only in physical mode. It is used to create a ring-like polygon. The number of segments used to approximate a circle can be altered with the **sides** command.

If the user presses and holds the **Shift** key after the center location is defined, and before the perimeter is defined by either lifting button 1 or pressing a second time, the current radius is held for x or y. The location of the **Shift** press defines whether x is held (pointer closer to the center y) or y is held (pointer closer to the center x). This allows elliptical donuts to be generated. This similarly applies when defining the outer radii, so that the inner and outer surfaces can have different elliptical aspect ratios, though the outer radius must be larger than the inner radius at all angles.

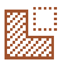
The **Ctrl** key also provides useful constraints. Pressing and holding the **Ctrl** key when defining the radii produces a radius defined by the pointer position projected on to the x or y axis (whichever is closer) defined from the center. Otherwise, off-axis snap points are allowed, which may lead to an unexpected radius on a fine grid.

When the command is expecting a mouse button press to define a radius, the value as defined by the mouse pointer (in microns) is printed in the lower left corner of the drawing window, or the X and Y values are printed if different. Pressing **Enter** will cause prompting for the value(s), in microns. If one number is given, a circular radius is accepted, however one can enter two numbers separated by space to set the X and Y radii separately.

While the command is active in physical mode, the cursor will snap to horizontal or vertical edges of existing objects in the layout if the edge is on-grid, when within two pixels. When snapped, a small dotted highlight box is displayed. This makes it much easier to create abutting objects when the grid snap spacing is very fine compared with the display scaling. This feature can be controlled from the **Edge Snapping** group in the **Snapping** page of the **Grid Setup** panel.

If the SpotSize variable is set to a positive value, or the `MfgGrid` has been given a positive value in the technology file, tiny round and donut figures are constructed somewhat differently. the figure is constructed somewhat differently. Objects created with the **round** and **donut** buttons will be constructed so that all vertices are placed at the center of a spot, and a minimum number of vertices will be used. The **sides** number is ignored. This applies only to figures with minimum radius 50 spots or smaller; the regular algorithm is used otherwise. An object with this preconditioning applied should translate exactly to the e-beam grid. See E.11 for more information.

7.7 The erase Button: Erase or Yank Geometry



Rectangular regions of polygons, boxes, and wires can be erased or "yanked" with the **erase** button. The user clicks twice or presses and drags to define the diagonal of the region to be erased. Selected objects are not erased. Wires maintain a constant width, and are cut at the points where the midpoint crosses the boundary of the erased area.

In physical mode, if the **Shift** key is held during the operation termination (click or button release), there is no erasure, however the pieces that would have been erased are "yanked", i.e., added to the yank buffer. The pieces are also added to the yank buffer when actually erased. The yank buffer chain has a depth of five, meaning that the contents of the last five yanks/erasures are available for placement with the **put** command.

Geometry in "foreign" windows can be yanked. These are physical-mode sub-windows showing a different cell than the current cell being edited (as showing in the main window). The foreign window is never erased (i.e., holding **Shift** is not necessary), but the structure that would be erased is added to the yank buffer. Thus, one can quickly copy a rectangular area of geometry from another cell into the current cell, by yanking with **erase** and placing with the **put** command (below **erase** in the side menu).

The **SpaceBar** toggles "clip mode". When clip mode is active, for objects that overlap the rectangle defined with the mouse, instead of erasing the interior of the rectangle as in the normal case, the material outside of the rectangle will be erased instead. The overlapping objects will be clipped to the rectangle. This applies whether erasing or yanking, again the yank buffer will acquire the pieces that would (or actually do) disappear in an erase operation.

When the **Ctrl** key is held before the box is defined, clicking on a subcell will cause the subcell's bounding box to be used as the rectangle. Thus, objects can be easily clipped to or around the subcell boundary. This applies when yanking as well. The standard erase is the inverse of the subcell paint operation in the **box** command.

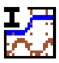
While the command is active in physical mode, the cursor will snap to horizontal or vertical edges of existing objects in the layout if the edge is on-grid, when within two pixels. When snapped, a small dotted highlight box is displayed. This makes it much easier to create abutting objects when the grid snap spacing is very fine compared with the display scaling. This feature can be controlled from the **Edge Snapping** group in the **Snapping** page of the **Grid Setup** panel.

The **box**, **erase**, and **xor** commands participate in a protocol that is handy on occasion.

Suppose that you want to erase an area, and you have zoomed in and clicked to define the anchor, then zoomed out or panned and clicked to finish the operation. Oops, the **box** command was active, not **erase**. One can press **Tab** to undo the unwanted new box, then press the **erase** button, and the **erase** command will have the same anchor point and will be showing the ghost box, so clicking once will finish the erase operation.

The anchor point is remembered, when switching directly between these three commands, and the command being exited is in the state where the anchor point is defined, and the ghost box is being displayed. One needs to press the command button in the side menu to switch commands. If **Esc** is pressed, or a non-participating command is entered, the anchor point will be lost.

7.8 The iplot Button: Interactive Analysis Plotting



The **iplot** command, available in electrical mode, is useful only if the *WRspice* program is available. Operation is similar to the **plot** button, whereby a command string is generated through selection of nodes and branches with the pointer. The command line can be edited in the usual way to generate, for example, functions of the plot variables. Pressing the **Enter** key saves the command. When the **iplot** button is active and a command has been saved, the plot is generated dynamically while a simulation, initiated with the **run** command, is in progress.

The **S** and **R** buttons, to the left of the prompt area, can be used to save and restore prompt line text in a set of internal registers.

Pressing the **iplot** button a second time will turn off the interactive plotting. Pressing **iplot** and then **Enter** will turn the interactive plotting back on. Of course, the trace points and plotting command can be modified before pressing **Enter**. In particular, if all prompt line text is deleted, pressing **Enter** will delete the internally saved command string, and turn interactive plotting off. Pressing the **iplot** button again will take as default text the string from the **plot** command, if any.

The command text and mark locations are saved with the cell data when written to disk, thus the **iplot** command is persistent.

7.9 The label Button: Create/Edit Labels

T

The **label** button is used to create or modify a text label. Labels are abstract annotation objects which do not appear in physical output. For physical text, use the **logo** command button.

If a label is selected before pressing the **label** button, then the selected label can be edited. Multiple labels can be selected, and each will receive the new label text. If more than one label is being changed, the command exits after the new text is entered on the prompt line, i.e., after **Enter** is pressed to terminate text entry.

If only one label is being changed, on pressing **Enter** the new text is "attached" to the mouse pointer, as for a new label. In this state, the text size, orientation, and justification can be changed as will be described below. The user can either click in a drawing window to place the label at the click location (effectively moving the selected label), or press **Enter** to update the selected label at the existing location.

This is the recommended way to change the size of a label: select it, press the **label** button, press **Enter** to keep the same text, adjust the size with the arrow keys, then press **Enter** again to update the label. This keeps the label in a standard size and aspect ratio which will match other labels. This would not be the case if the **Stretch** command or operation was used instead.

If no label was initially selected, after the label text has been entered, the label will appear ghost-drawn, attached to the mouse pointer. The text will be rotated or mirrored according to the current transform, as set from the pop-up provided by the **xform** button in the side menu. Instances of the label are placed where the user clicks in a drawing window.

Label text in entered in the prompt line. While editing, if the user clicks on an existing label in a drawing widow which is contained in the current cell, the text of that label will be inserted at the prompt line cursor. Hypertext entries (see refhypertext) in the label will be preserved. If the existing label is a "long text" label (described below), the long text attribute will be lost, unless the prompt line is empty before clicking on the label. Particularly in electrical mode, clicking on other objects in a drawing window will insert text at the cursor position, as will be described. Pressing **Enter** terminates the label text and will allow placement of copies of the new label.

The size and justification of the label can be adjusted with the arrow keys, before it is placed. The arrow keys have the following effect:

Up	enlarge by 2
Right	enlarge by 10%
Down	reduce by 2
Left	reduce by 10%

The initial size of a label is determined by the present default label height, and the magnification of the current drawing window. The default label height is 1.0 microns, which can be reset by setting the LabelDefHeight variable to a different value. The default height is the smallest size available through scaling with the arrow keys. Generally, *Xic* functions that create new labels will use the default label height. The default height of one micron is too large for modern semiconductor processes, so one should redefine LabelDefHeight in the technology file to a more suitable value, typically the minimum feature size.

By default, the label is anchored at the lower left corner, though this justification can be changed by holding the **Shift** key while pressing the arrow keys. The **Left** and **Right** arrows cycle through left, center, and right justification. The **Up** and **Down** arrow keys cycle through bottom, center, and top justification. Finally, holding the **Ctrl** key while pressing the arrow keys will change the current rotation angle. The arrow keys implicitly cycle through the angle choices, with **Up** and **Right** cycling in the opposite sense from **Down** and **Left**.

Labels are scalable, and can be stretched with the **Stretch** button in the **Edit Menu** or with button 1 operations.

Newlines can be embedded in the label text by pressing **Shift-Enter**. The displayed label will contain line breaks at those points. The justification applies to the block, and line-by-line within the block.

Labels are shown in legible orientation (i.e., left to right or down to up) by default, independent of the actual transformation. If the **Label True Orient** button in the **Main Window** sub-menu of the **Attributes Menu** or the sub-window **Attributes** menu is set accordingly, labels will be shown in their actual orientation.

Pressing the **Delete** key after the label text has been entered will repeat prompting for new label text. Labels have fixed size as compared with layout geometry.

7.9.1 Device Property Labels

Labels are created internally for device properties in electrical mode. These labels can be moved, deleted, and edited just as user-supplied labels. Once deleted, though, such labels can not be recreated except by recreating the device, or by using the **!regen** command. The underlying property is not deleted, it simply is not displayed in a label.

These labels can be “hidden” by clicking on the label text with button 1 with the **Shift** key held. This replaces the label text with a small box icon. Shift-clicking the icon will redisplay the text. This can be useful when long labels obscure other features. See 7.9.7 for more information.

Labels can be edited by selecting the label before pressing the **label** button. If the label was generated for a property in electrical mode, the underlying property is also changed. This is a quick way to modify device properties, without invoking the **Properties** command button in the **Edit Menu**.

7.9.2 Wire Net Name Labels

Similar to the property labels, electrical wires that participate in schematic connectivity can have a bound label that provides a name for the net containing the wire.

Unlike the device labels, wire net labels are created by the user. If the **label** command is started with a single selected wire on an electrically-active layer, the label created will be bound to that wire. Thus, to create a label for a wire, select the wire, press the **label** button in the side menu, and create the label. These labels can exist on any layer.

Once created, these labels can be edited in the same manner as property labels, i.e., select the label and enter the label command by pressing the side menu **label** button.

7.9.3 Ctrl-a and Ctrl-p

In electrical mode, outside of any command, pressing **Ctrl-a** will cause the associated labels of any selected device or wire to also become selected. If labels are selected, then pressing **Ctrl-a** will cause their associated device or wire to also become selected. The associated labels can be deselected by pressing **Ctrl-p**. This is useful for determining which labels are associated with a given device or wire, and *vice-versa*.

7.9.4 Spicetext Labels

In electrical mode, for efficiency reasons it is best not to use the SCED layer for labels. If the current layer is the SCED layer, a new label will instead be created using the ETC1 layer. If for some reason a label is required on the SCED layer, the **Change Layer** command in the **Modify Menu** can be used to move an existing label to the SCED layer.

In electrical mode, labels can be used to enter arbitrary text into the SPICE output. There are two methods to achieve this. In addition, the SpiceInclude variable can be used to add a file inclusion to the SPICE output.

If an electrical layer named “SPTX” exists, labels on this layer will be included, verbatim, as separate lines in SPICE output, unless the label is a “spicetext” label (below). These labels are sorted by position, top-to-bottom and left-to-right in output, and are placed ahead of the spicetext labels. A label on the SPTX layer in the format of a spicetext label will be output as a spicetext label.

If the first word of the label is of the form

`spicetext`N

the label is a “spicetext” label, and the text which follows will be entered verbatim as a separate line in the SPICE output. The spicetext labels can appear on any layer. The integer N, which is optional, is a sorting parameter. If there are multiple labels containing SPICE text, they will be sorted by N before being added to the SPICE output. Smaller N will appear earlier in the listing, with omitted N corresponding to a value of zero. The `spicetext` lines are written as a contiguous block in the listing.

Any text which can be interpreted by the SPICE simulator in use can be added using these methods, but erroneous syntax will of course cause errors as the SPICE text is sourced.

7.9.5 “Long Text” Capability

When editing or creating unbound labels, or labels for physical or certain electrical properties (value, param, and other), there is provision for entering a block of text that will not be visible in the layout or schematic. This avoids cluttering the screen with labels containing large blocks of text. Rather, a symbolic form will be shown instead of the full text.

This same capability applies when adding or editing properties from the **Property Editor** provided by the **Properties** button in the **Edit Menu**.

This capability is useful for properties which require a large block of text, such as a long PWL statement in a value property for SPICE. It is not possible to edit a large text block in the prompt area, and if displayed would cause the screen to be obscured or cluttered. The full text is added to SPICE output, however, and is available as the property value in functions that query the value.

It is also useful for the `spicetext` labels, so that a block of text can be inserted into SPICE output, rather than one line. Remember that the text entered into the window must begin with “`spicetext`” and an optional integer, for the text to appear in SPICE output.

When entering a label where this “long text” capability applies, a small “**L**” button will appear to the left of the prompt line, and this will be active when the text cursor is in the leftmost column. Pressing this button will set the internal flag for "long text", and open the text editor window for the text. Any text that was previously entered in the prompt line will be added to the text editor window, or, if the label was already in long text mode, the existing text will be shown in the editor.

If preexisting text was present on the prompt line when the **L** button was pressed, that text will be loaded into the text editor, but any hypertext entries will be converted to plain text. The long text blocks do not support the hypertext feature.

Pressing **Ctrl-t** has the same effect as pressing the **L** button when the button is visible and active.

From the editor window, one can edit the block of text, then press **Save** in the editor's **File** menu to complete the operation, or **Quit** to abort. The on-screen label will simply say "`[text]`" for a normal "long text" property or non-associated label, or have the standard form for a script label (described below);

The long text labels can be edited with the label editor, as can normal labels, by selecting the label before pressing the **label** button. The prompt line will display "`[text]`" as a hypertext entry. Pressing **Enter** or the **L** button will bring up the text editor loaded with the text associated with the label, allowing editing.

To convert a long text label to a normal label, instead of bringing up the text editor, the hypertext "`[text]`" entry can be deleted in the prompt line. Deleting the entry will place as much of the text block as possible on the prompt line, and delete the text block and the association of the label or property as a long text object.

7.9.6 Script Labels

Xic provides the ability to embed a script or script reference in a label, which is executed when the user clicks on the label. These are created like any other label, but have the form

`!!script [name=`*word*`] [path=`*path*`] [`*script text*`...]`

The leading token in the label must be "`!!script`" to indicate that the label text is executable. This is followed by zero or more keyword/value pairs as shown, followed by the script text that will be executed. The keywords and values must be separated by '=' with no space. The value is a single token, which should be double-quoted if it contains white space. These are optional.

The keywords have the following interpretations.

`name=`*some_word*
: The script label is rendered on-screen as *some_word* surrounded by a box. If no name is given, the word "`script`"" is shown.

`path=`*some_path*
: If this is given, then the script to be executed is given by *some_path* and any executable statements in the label are ignored. The *some_path* can be an absolute path to a script file, or can be the name of a script file expected to be found in the script search path.

Any remaining text is executed as script commands, if `path` is not given. For short scripts, semicolons can be used as command terminators in a single line. Otherwise, a text editor can be invoked on the label string by pressing the "L" (long text) button when creating the label.

Clicking on a script label will execute the script, and not select the label as with normal labels. To select a script label, hold **Shift** while clicking on the label, or drag over the label (area select). If a script label is selected, it will not execute when clicked on, but rather be deselected.

For example, suppose that a user has a large layout, with a small section that the user often needs to zoom into. The user can create a script label to perform the zoom operation. After zooming in, one can note the position and estimate the width of the drawing window. Then, one would create a label such as

`!!script name=zoom Window(`*x*`, `*y*`, `*width*`, GetWindow())`

and place it somewhere convenient. The x, y, and *width* above of course represent the actual values (in microns). Clicking on the label will always zoom to this area.

7.9.7 Label Size Issues

In electrical mode, property text labels can be displayed or "hidden". If a label is hidden, the text is not displayed, only a small box at the text reference point is shown.

Labels with text size longer than a certain length will be shown as hidden by default. Hidden labels can be made visible, and *vice-versa* by clicking on the label or small box with the **Shift** key held. The label text can also be shrunk (with the **Stretch** command in the **Modify Menu** or with button 1 operations) to make it visible.

The label hidden status is persistent when the cell is saved in any format, however changing the display status does **not** change the modified state of a cell, thus this can be done in IMMUTABLE cells.

It should be noted that the "real" bounding box of the label, which is used to set the cell bounding box, is always the bounding box of the actual text. The hidden display mode is only available for the labels that contain property strings in electrical mode. Hidden labels can be selected only over the small box, and only the small box is highlighted. However, when moving or stretching, the entire "real" bounding box is highlighted.

The size threshold can be changed with the **Maximum displayed property label length** entry in the **Window Attributes** panel from the **Set Attributes** button in the **Attributes Menu**. Equivalently, the variable LabelMaxLen can be set to an integer greater than 6 with the **!set** command. The units are the width of a default-size character cell. In releases prior to 2.5.66, the default length was 32 default character size cells. In this and later releases, the value is 256 character cells. The larger threshold makes the nondisplay of label text much less probable, as this feature has been confusing to users.

Another way to obscure a long label is to convert it to a "long text" label.

To "hide" a label using the "long text" feature:

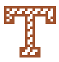
1. Select the label.
2. Press the side menu **label** button (with the black 'T' icon).
3. Press the gray **L** button that appears to the left of the prompt line. This will cause the text editor to appear, loaded with the label text. If there is no **L** button, then the property can't use long text, which is true for properties that are "always" short, such as for device and model names.
4. In the text editor, press **Save** in the **File** menu. The editor will disappear, and the label displayed on-screen will have changed to "`[text]`".

To convert back to a normal label:

1. Select the long-text label (“`[text]`”).
2. Press the side menu **label** button (with the black ’T’ icon).
3. With the cursor under “`[text]`” on the prompt line, press the **Delete** key. The full label text will appear on the prompt line.
4. Press **Enter**. The label will be shown normally.

Long property text labels can also be broken into multiple lines by adding embedded returns. These are added with **Shift**-**Enter** while the string is being edited. Note that this generates newlines in the SPICE output, so that in most cases the extra lines should begin with the “+” continuation character.

7.10 The logo Button: Create Physical Text



The **logo** command allows the creation of physical text and images for labeling, identification, etc. Operation is similar to the **label** command, where the arrow keys alter text or image size, **Shift**-arrow cycles through the justification choices, and **Ctrl**-arrow cycles through the rotation angles. By default, the text is implemented with rounded-end wires in the current layer, using a vector font.

For rendering text, there are three font possibilities. The default font is a vector font which constructs the characters using wire objects. The Manhattan font is a built-in bitmap font from which the characters are constructed using Manhattan polygons. The Manhattan font is fixed-pitch with an 8X16 map. The “pretty” font is one of the system fonts, which similarly creates characters constructed as Manhattan polygons. Logic is applied to extract the “best” rendition from anti-aliased fonts, which do not have a precisely defined shape. Some fonts may look better than others in this application.

While in the **logo** command and using the vector font, pressing the **Ctrl**-**Shift**-arrow key combinations will adjust the path width; the **Up** and **Right** arrow keys increase the width, **Down** and **Left** arrows decrease the path width.

The LogoPathWidth variable tracks the current path width setting. The LogoEndStyle variable tracks the current end style setting.

Instead of a text label, the **logo** command can be used to place an image. The image must be provided by a file in the XPM format. This is a simple ASCII bitmap format, commonly used in conjunction with the X-windows system on Unix machines. Other types of bitmap files can be converted to XPM format with widely available free software, such as the ImageMagick package. Several XPM files are supplied in the help directory for *Xic* (located by default in `/usr/local/xictools/xic/help`), which illustrate the format.

This feature is enabled in the **logo** command by giving the path of an XPM file, which must have a “`.xpm`” suffix, as the text string. This will cause the image to be imported such that it can be scaled, transformed, and placed, just like a normal label. The background color (the first color listed in the XPM file) is taken as transparent. All other layers found in the XPM file are mapped to the current layer. The image is rendered as a collection of Manhattan polygons.

Unlike in releases 3.0.11 and earlier, there is no attempt to limit feature sizes according to design rules. The minimum size of a character is set by the internal resolution, while the maximum size is about

.4 X .7 cm. Once the text is entered, the size and other attributes can be changed with the arrow keys, and the text is placed where the user clicks in the drawing with button 1. The text can be reentered, i.e., a new label or image file defined, if the **Delete** key is pressed.

Alternatively, a fixed "pixel" size can be specified. In this case, the arrow keys will pan the display window, and have no effect on the label or image size.

The default operation is to apply the text or image feature directly in the current cell, where the user clicks. It is also possible to create a subcell containing the text, which is instantiated at the clicked-on locations. This may be more efficient if there are many copies of the same label.

Note that use of the vector font may produce design rule violations, which are pretty much inevitable due to the presence of acute angles in some characters. Use of the other fonts, which are rendered using Manhattan polygons, can avoid design rule violations, if the "pixel" size is larger than the MinWidth and MinSpace design rules for the layer. When physical text (or an image) is placed with the **logo** command, interactive design rule checking is suppressed. The NoDRC flag can be set on the new label, or the NDRC layer can be used, to permanently suppress DRC.

It is possible to change the font used for the **logo** command. The default font is set internally by *Xic*, however individual characters or the whole font will be updated upon startup if a file named "`xic_logofont`" is found along the library search path, which contains alternative character specifications.

7.10.1 The Logo Font Setup Panel

While the **logo** command is active, the **Logo Font Setup** panel is visible, though this can be dismissed without leaving the **logo** command. The top of the panel provides three "radio" buttons for selecting the font: **Vector**, **Manhattan**, and **Pretty**. The LogoAltFont variable tracks the choice in these buttons.

Below the **Font** choice buttons is the **Define "pixel" size** check box and numeric entry window. When checked, the numeric entry area is enabled, and the value represents the size of a "pixel" used for rendering the label or image, in microns. When checked, the arrow keys have no effect on label or image size, instead they revert to their normal function of panning the display window. This feature is tied to the LogoPixelSize variable, which when set to a real number larger than 0 and less than or equal to 100.0 will define the "pixel" size used in the **logo** command.

There are two option menus in the **Logo Font Setup** panel which set the end style and path width assumed in the wires used for constructing characters with the vector font. The user can set these according to personal preference. Although rounded end paths may look better, they are somewhat less efficient in terms of storage and processing, and are not handled uniformly (or at all) in some CAD environments. For example, rounded-end wires may be converted to square ends when written as OASIS data.

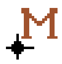
The **Select Pretty Font** button brings up the **Font Selection** panel, allowing the user to select a system font for use as the "pretty" font. In the **Font Selection** panel, the user can select a font, then press the **Set Pretty Font** button to actually export the choice. This will set the LogoPrettyFont variable.

The **Create cell for text** check box, when checked, sets a mode where new labels and images are instantiated as subcells rather than directly as geometrical objects. In addition to generating a master cell in memory, a native cell file with the same name is written in the current directory. The boolean variable LogoToFile tracks this state of this check box.

The name of the file used for the label is internally generated, and is guaranteed to be unique in the current search path. The name consists of the first 8 characters of the label, followed by an encoding of the various parameters related to the label. For a given label, the uniqueness of the file name prevents recreating the same label file in a subsequent session.

The **Dump Vector Font** button will create a file containing the vector font (see C.1) currently being used by the **logo** command. By default, the vector font uses the same character maps as the vector font used to render label text on-screen. However, these maps can be overridden by definitions from a file. The **Dump Vector Font** button can be used to dump the current set of character maps to a file. Character maps from this file can be modified and placed in a file named "`xic_logofont`" in the library search path, in which case they will override the internal definitions when producing vector-based characters in the **logo** command.

7.11 The nodmp Button: Node (Net) Name Assignments



The **nodmp** button, which is available in the electrical mode side menu, will bring up the **Node (Net) Name Mapping** panel which is used to display and alter the names used for "nodes" (single-conductor wire nets) in the schematic, and in SPICE and other output. This name may be internally generated, or may be derived from a terminal name, or may be assigned by the user. This panel is also brought up by the **Find Terminal** button in the **Setup** page of the **Extraction Setup** panel, which is obtained from the **Setup** button in the **Extract Menu**.

First, to facilitate the discussion that follows, some terminology will be introduced. See also the section on wire net naming in 4.2.8.

scalar
: Single-conductor wire nets, or "nodes" (from SPICE terminology) are referred to as "scalar" nets. These are the actual circuit connections, which are compared in layout vs. schematic (LVS) testing. *Xic* also allows multi-conductor (including single-conductor) "vector" and "bundle" nets. These actually reference and organize the nodes, but do not provide actually connectivity, except through name matching. The present **Node (Net) Name Mapping** panel applies only to the scalar nets.

associated name
: A scalar wire net, or "node" may have "associated names". These names derive from named terminal devices which may be connected to the net, or from labeled wires which are connected to the net. Both the terminal device and the labeled wire derive the net name from the text of an associated label. The labels can be edited, which will change the text of the associated name. A net may have any number of associated names.

cell terminal name
: Every electrical contact point of a cell has a name. This name was assigned when the cell terminal was created with the **subct** command button in the side menu, or if no name was given a default name is used.

  It is also possible to name cell contact terminals from the **Edit Terminals** command button in the **Setup** page of the **Extraction Setup** panel. This panel is brought up with the **Setup** button in the **Extract Menu**.

global names
: Certain names are registered within *Xic* as "global names", and are kept in an internal string table.

These names are known at every level of the cell hierarchy. There is always at least one global name defined, the ground node with name "`0`".

Global names are easily created by the user, as any node name ending with an exclamation point ('`!`') is taken as a global name. For example, "`vdd!`" would be taken as a global name.

Global node names are also set with the DefaultNode global properties, in the device library file. They may be used as default nodes in some devices. In particular, the "three terminal" **nmos** and **pmos** devices included in the default library make use of this feature, defining global node names "`NSUB`" and "`PSUB`" that connect to the device substrate.

assigned name
: Names that are specified from the **Node (Net) Name Mapping** panel using the **Map Name** button will be referred to as "assigned names".

A wire net can clearly have a number of names associated with it. The actual name for the node will be chosen according to the priorities listed below.

1. If a net has an associated name that matches a global name, that global name is used, and this can not be overridden by the user.

   If two or more global names match associated names in the net, the name chosen will be the one earliest in ASCII lexicographic order. This situation is unlikely and probably represents a topology error.

2. If a net is given an assigned name, that name will be used.

3. If a net contains a cell terminal, the cell terminal name will be used. It is possible that more than one cell terminal is connected to the node, in which case the name chosen will be the one earliest in ASCII lexicographic order.

4. If the net has an associated name, that name will be used. It is possible that more than one associated name will be found, in which case the name chosen will be the one earliest in ASCII lexicographic order.

5. The net will be given a name based on the internally-generated node number.

For names other than the internally generated node numbers, the name is predictable. The internally generated numbers will change if the circuit is modified, or possibly for other reasons. Thus, if netlist or SPICE output is to be used in another application, it may be important to assign names to nodes to be referenced by name.

The **Node (Net) Name Mapping** panel contains two text listing windows. The left (node listing) window lists all of the nets in the current cell schematic. An entry in the list can be selected by clicking on the text with the mouse. When a net is selected in this list, the terminals to which the net connects are listed by name in the right (terminal listing) window. Entries in the terminal listing can be selected as well by clicking on the text with the mouse. In either window, the selected entry, if any, is highlighted.

There is a "grip" in the region separating the two text listings, which can be dragged horizontally to change the relative widths of the windows.

The left column in the node listing contains the internal node numbers, which can change arbitrarily if the circuit is modified. Entries in the second column are the mapped names, i.e., the names used in SPICE and netlist files. If the second column entry is blank, no name could be found for the net, and *Xic* will create a name from the node number for use in output. The third column will contain the letter

"Y" if the node has a name assigned by the user, and/or a "G" if the node name is that of a global node (including ground). Both letters will appear if the user assigns a name that matches a global name, which includes any name that ends with an exclamation point. The "G" nodes without Y can not be renamed by the user.

When a node is selected in the left text window, the right text window lists terminals and other features that are found in the selected net. This includes

- Device and subcircuit instance terminals.
- Named terminal devices. These start with a 'T' character, followed by a space, followed by the name from the terminal label.
- Named wires. These start with 'W' followed by space and the name from the wire label.
- Cell contact terminal names.

The names used for device terminals are a concatenation of the device name and the terminal names as supplied in the node properties in the device library file, if a name was given. If no name was given, a default name is constructed as *devicename_contactnum*. That is, the device name, followed by an underscore, followed by an internal index number for contacts of that device. The device name starts with a letter which is the SPICE key for that device type. Subcircuits are similar, and the terminal names begin with 'X'.

In the electrical schematic drawing, when a net is selected in the node listing window, wire objects that are included in the selected net are highlighted. Each of the device and subcircuit instance terminals listed in the terminal listing area will have a small highlighting box drawn around its location. If one of the terminals in the terminal listing is selected, that terminal will be displayed using highlighting.

The panel will cooperate closely with the physical extraction system when the **Use Extract** check box is checked. This means that extraction/association will be performed as needed so that terminal locations are correctly defined in the physical layout as well. In this case, a terminal selected in the terminal list will be shown in physical layout windows, as well as the schematic. If the check box is not checked, extraction data will be used if present when showing the terminal in layouts, but there is no attempt to maintain currency. The **Node (Net) Name Mapping** panel is also available from the **Find Terminal** button in the **Extraction Setup** panel in both physical and electrical modes, in addition to the side-menu button in electrical mode.

When an entry in the terminal listing window is selected, the **Find** button, below the listing, is un-grayed. Pressing the **Find** button will bring up a sub-window displaying the current cell, with the selected terminal at the exact center of the display. One can press the numeric keypad **+** key repeatedly to zoom in to the terminal location, and the terminal will remain centered. Further, if **Use Extract** is set or the extraction state is current, the terminal will also be displayed and centered if the sub-window is changed to physical mode.

When the **Click-Select Mode** button is pressed, a command state is entered whereby clicking on a wire or contact point in a drawing window will select that net. This works a bit differently depending on the state of the **Use Extract** check box. If the box is checked, the button will bring up the **Path Selection Control** panel from the extraction system. This allows selection of conducting paths in the layout windows by clicking on objects. The corresponding net will be selected in the node listing window, with corresponding highlighting shown in schematic windows. One can also click on wires and terminal locations in the schematic, and the clicked-on net will become selected. The corresponding conductor group will be displayed highlighted in layout windows.

With **Use Extract** not checked, the **Path Selection Control** panel will not appear, but clicking in schematic windows will have similar effect. The system will once again use extraction data if available to map button presses in layout windows to a conductor group and back to the coresponding electrical net to be highlighted. However, there is no highlighting of the physical conductor group.

In either case, the clicked-on node will be shown selected in the node listing window, and scrolled into view if necessary. The terminal listing window will show the selected net details as usual. **Click-Select Mode** is exited if another command is started, or **Esc** is pressed, or the **Click-Select Mode** button is pressed again, or, with **Use Extract** checked, the **Path Selection Control** panel is retired by any means.

The **Deselect** button will deselect selections in the node listing window, and the corresponding highlighting in the drawing windows. The terminal listing window becomes empty.

It is also possible to search for nets and terminals by name using the controls just above the two listing windows. The two "radio" buttons select whether to search for node or terminal names. One enters a "regular expression" into the text area. The button to the left of the text entry initiates the search. A matching net is selected as is the matching terminal if searching for terminals. One can press the button again to move to the next and subsequent matches. If there is no initial selection, perhaps because **Deselect** was pressed, the search area starts at the top and extends toward the end of the listing. If a net is selected, the search starts with the next item (terminal or net) after the selection end extends toward the end.

The regular expression conforms to POSIX.1-2001 as an extended, case-ignored regular expression. On a Linux system, "`man grep`" provides a good overview of regular expression syntax and capability. However, one probably doesn't need to know much more than

1. A given string will match any name containing the string, case insensitive.
2. The carat ('`^`') character matches the beginning of a name.
3. The dollar sign ('`$`') character matches the end of a name.

If the third column in the node listing window is not '`G`', then an overriding name for the selected node can be assigned with the **Map Name** button, but only while in electrical mode. To apply a name, select a node in the node listing area, then press the **Map Name** button. A new name will be prompted for in a pop-up window. The name can be any text token (white space is not allowed), however it is up to the user to ensure that the name makes sense in the context of the output. For example, for SPICE output, the node names must adhere to the rules for valid node names in SPICE. After pressing **Apply**, the second column in the listing will be updated to show the new name, and the third column will show "`Y`". Again, this can only be done while in electrical mode, in physical mode the button remains grayed.

The node naming can actually modify circuit topology, which can be a powerful feature or a curse. If two nets share a name, they will be merged, and the left window will reflect this. Thus, it is easy to make connections using node name mapping that are not obvious when looking at the schematic. For this reason, if the user is about to apply a duplicate name, a confirmation pop-up will appear. The user is given the choice to back out of the operation, or continue.

The node name assignment works by association with a connection point in the net, equivalent to a hypertext reference. This association persists if the object is moved, and is transferred to another device or wire if the object is deleted, if possible. In some cases it may get lost, however, so an assigned name may have to be reentered after the circuit is edited.

In electrical mode, an assigned name can be deleted by first selecting the node in the node listing area, then pressing the **Unmap** button. The **Unmap** button is un-grayed only if the third column of the selected node shows "Y" indicating that it has an assigned name. On pressing the button, the name will revert to the default name. This may effectively change circuit topology by undoing the net merging brought about through net name assignments. Again, this operation is available only in electrical mode.

The internal data structure representing node name mapping, and the listings, will be in one of two states. Either devices and subcircuits with the nophys property will be included as normal devices and subcircuits, or these will be ignored. In the latter case, if the nophys property has the "shorted" option, the terminals will be effectively shorted together, which will obviously change the node numbering.

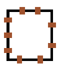
The current state is as set by the last function to generate the connectivity map. Functions in the extraction system will always recognize the **nophys** properties, and build the map excluding these devices but taking the "shorted" nophys devices as shorted. Then, the schematic will correspond to the actual physical layout. Functions in the side menu which generate a SPICE listing will ignore nophys properties and include all such devices in the net list. This produces a schematic appropriate for SPICE simulation.

The **Use nophys** button is used to switch between these two representations, and the state of the button will be reset if another function changes the state.

When the **Use nophys** button is pressed, devices and subcircuits with the nophys property set will be *included* in the listings, just as "normal" devices. Their terminals will be listed in the terminals listing window. The nophys property is ignored in this case, as will be true when a listing is being prepared for SPICE output from functions in the side menu.

When the **Use nophys** button is not pressed, devices and subcircuits with the nophys property will be ignored in the listings, and the node numbering will respect the "shorted" nophys properties. Terminals from these devices and subcircuits will not be listed in the terminal listing window. This mode is consistent with the usage by the extraction system.

7.12 The Place Button: Cell Placement Control Panel



The **place** button in the side menu brings up the **Cell Placement Control** panel which allows instances of cells (subcells) to be added to the current editing cell.

When the **Place** button in the panel or the **place** button in the side menu is active (the two buttons show the same status), the current master can be instantiated at locations where the user clicks ("place mode"). The bounding box of the cell is ghost-drawn and attached to the pointer. The orientation and size of the instance are set by the current transform. If the **Cell Placement Control** panel is dismissed the place mode, if active, is exited. The place mode can be exited with the **Esc** key, or by pressing the **Place** button (either one) a second time. The panel is not popped down when place mode is exited.

The substructure of cell instances being placed is highlighted to the depth shown in the main window. This facilitates alignment with other objects. One can change the display depth to reveal more or less of the substructure.

From the **Open** command in the **File Menu**, if one holds down **Shift** while selecting one of cells from the history list, the **Cell Placement Control** panel will appear with that cell added as the current master. This applies to cell names and not the "**new**" entry. This is a quick backdoor for instantiating cells recently edited.

In electrical mode, when a connection point of a device or subcell is near another connection point, it will snap to that location and a small dotted box will be drawn around the point. This facilitates placement of devices and subcircuits in the schematic. While the **Shift** or **Ctrl** keys are held, this feature is disabled.

Cells can be placed individually, or as arrays in physical mode. When the **Use Array** button is active, cells will be placed as arrays, governed by the currently set array parameters. The array parameters can be entered into the four text fields below, only when the **Use Array** button is active. Arrays are allowed in physical mode only. If this button is not active, single cells are placed.

The array replication factors Nx and Ny can be set to any value in the range of 1 through 32767. The upper limit is set by the GDSII file format, and is enforced by *Xic* to avoid portability problems.

The spacing values Dx and Dy are edge to adjacent edge spacing, i.e., when zero the elements will abut. If Dx or Dy is given the negative cell width or height, so that all elements appear at the same location, the corresponding Nx or Ny is taken as 1. Otherwise, there is no restriction on Dx or Dy except that very large (unphysical) values can cause integer overflow internally.

The **!array** command can be used to convert existing instances into arrays, and to modify the array parameters of existing arrays.

In physical mode, the reference point of the cell, which is the point in the cell located at the pointer, can be set to either the cell's origin, or to one of the cell's corners. A drop-down menu in the **Cell Placement Control** panel indicates the present selection, and allows the user to make a new choice. The nomenclature "Upper Left", etc., refers to the corner of the untransformed cell array bounding box. When place mode is active, pressing the **Enter** key repeatedly will cycle the reference point around the corners and back to the origin.

In electrical mode, the cell reference point is always set to the location of the reference terminal, which is usually the first terminal defined. If the cell has no terminals, the reference point can be cycled around the corners, as in physical mode, however for corners the reference point is snapped to the nearest grid location. This should prevent device terminals from being located off-grid. An electrical cell should always have terminals (assigned with the **subct** command in the electrical side menu) if it is to be part of the circuit, and not some kind of decoration or annotation.

When the **Smash** button is active, is active, instances will be smashed into the parent where the user clicks, meaning that the cell content will be merged into the parent cell, rather than creating a new instance. The flattening is one-level, so that any subcells of the cell being placed become subcells in the parent.

When the **Replace** button is active, existing cells are replaced with the new master when clicked on. and no cells are placed if the user clicks in the area outside of any subcells. When a cell is replaced, the placement of the new cell is determined in physical mode by the setting of the reference selection drop-down menu. For example, if this setting is "Upper Right", the new cell untransformed upper-right corner will be placed at the existing cell untransformed upper right corner.

In electrical mode, the reference terminal (the first connection point) is always placed at the same location as the reference terminal of the replaced cell. In either case, any currently active transformations are performed in addition to the transformations of the replaced cell on the new cell.

Cells can be placed or replaced only when place mode is active, i.e., when the **Place** button in the **Cell Placement Control** pop-up or the **place** button in the side menu is active.

When place mode becomes active, the current transform is cleared. The current transform is saved in Register 0 and cleared when switching between cells to place or upon pressing the **Desel** button. Pressing the **Forward Slash** button swaps the current and saved transforms.

If the **Use Array** button is active when cells are being replaced, the replaced cell will take the array parameters from the **Cell Placement Control** panel. Otherwise, the array parameters are unchanged during replacement. Note that it is possible to replace an instance with another instance of the same cell, but with different array parameters. This is one way that array parameters can be "edited".

The **Dismiss** button will retire the **Cell Placement Control** panel, and exit place mode.

The cell currently being placed, the "master", can be selected in several ways. A list of masters is kept, and can be viewed with the menu button in the **Cell Placement Control** panel. Pressing and holding button 1 with the pointer on the menu button issues a drop-down menu, whose entries are highlighted as the pointer passes over them. A selection is made by releasing button 1 over one of the selections. Pressing the **New** button in this menu brings up a dialog box which allows the user to enter a new master name.

The pop-up list of cells will grow with each addition until a limit is reached, at which point new entries will replace the oldest one. The **New** item is always at the top of the list. The list consists of the most recent masters specified, either with the **New** button, or through the **Place** button in the **Cells Listing** or **Files Listing** panels.

The maximum number of masters saved in the menu can be specified with the **Maximum menu length** entry area just below the menu. The default is 25, which may not be suitable for some screen resolutions or window systems. It is not very useful if the pull-down menu extends off-screen. This entry tracks the value of the MasterMenuLength variable. The variable can be set as an integer or cleared to change the value, which is equivalent to changing the integer entry in this panel.

When a new entry is selected, a dialog pop-up appears for the new cell name. If a selection can be found in the various panels that provide file or cell selection, that selection is pre-loaded into the dialog as a default. Each of these sources is tested in order, and the first one that is visible and has a selection will yield the default cell name.

- A selection in the **File Selection** pop-up from the **File Select** button in the **File Menu**.
- A selection in the **Cells Listing** pop-up from the **Cells List** button in the **Cell Menu**.
- A selection in the **Files Listing** pop-up from the **Files List** button in the **File Menu**, or its **Content List**.
- A selection in the **Content List** of the **Libraries** pop-up from the **Libraries List** button in the **File Menu**.
- A selection in the **Cell Hierarchy** pop-up from the **Show Tree** button in the **Cell Menu** or from the **Tree** button in the **Cells Listing** pop-up.
- A cell name that is selected in the **Info** pop-up, from the **Info** button in either the **View Menu** or the **Cells Listing** pop-up.
- The name of a selected subcell in the drawing window, the most recently selected if there is more than one.

The first time the **Cell Placement Control** panel comes up, the user is prompted for the name of a cell, just as if the **New** menu button was pressed.

The name provided can be a file containing data in one of the supported archive formats, the name of an *Xic* cell, or a library file. If the name of an archive file is given, the name of the cell to open can follow the file name separated by space. If no cell name is given, the top level cell (the one not used as a subcell by any other cells in the file) is the one opened for placement. If there is more than one top level cell, the user is presented with a pop-up choice menu and asked to make a selection. If the file is a library file, the second argument can be given, and it should be one of the reference names from the library, or the name of a cell defined in the library. If no second name is given, a pop-up listing the library contents will appear, allowing the user to select a reference or cell.

The given given string can also consist of the name of a Cell Hierarchy Digest (CHD) in memory, optionally followed by the name of a cell known within the CHD hierarchy. If no cell name is provided, the cell name configured into the CHD is understood. The string can also contain the name of a saved CHD file, with an optional following cell name.

The **Cell Placement Control** panel is sensitive as a drop receiver. If a file name is dragged over the panel and the mouse button released, the behavior is as if the **New** button in the masters menu was pressed, and the file name will be loaded into the dialog window.

7.13 The plot Button: Generate SPICE Plot

The **plot** button, available only in electrical mode, gathers input for plotting via *WRspice*.

The prompt area displays the command string as it is being built. Clicking on nodes or device "hidden" targets (usually indicated by a '+' symbol in the device schematic representation) will add hypertext entries to the command string, and will add a marker to the screen at the clicked-on location. One can click anywhere on a wire, or on subcircuit and device connection points to select nodes. Clicking on a marker will delete the marker, and the corresponding entry from the string.

Some devices have "hidden" nodes for accessing internal variables for plotting, such as current through a voltage source or the phase of a Josephson junction. By convention, these are indicated with a '+' mark in the symbol. For many devices, this will access the current through the device. The marker for a current has an orientation in the direction of positive current flow. Ordinary node markers have no orientation, and access the node voltage.

One can click on reference points to any depth in the hierarchy, though selection requires that the cell be showing as a schematic, and as expanded. To make selections inside a subcircuit that is shown as a symbol, one can use proxy windows (see 3.1.3). Holding down both the **Shift** and **Ctrl** keys, and clicking on a subcircuit instance, will bring up a sub-window displaying the master of the clicked-on instance in schematic form. One can click on objects in this window in the normal way, and plot points will be added to the prompt line.

Holding the **Shift** key while clicking on a device of subcircuit not over any node or "hidden" location will enter the hypertext device or subcircuit name. These are not often needed in plot command strings, and the requirement of holding down **Shift** prevents unwanted returns.

Markers can be deleted by clicking on them, or by deleting the corresponding hypertext in the prompt line. Multiple markers can reference the same node, as long as they are spatially distinct.

Existing marks can be dragged and dropped to a new location, which must also reference a node or reference point, as for the initial placement. If dropped on an existing plot mark, the two marks will exchange locations, both as marks in the drawing window, and hypertext entries in the prompt line.

The prompt line text is equivalent to the string given to the **plot** command in *WRspice*. The string can be edited in the usual way. The user can add text to combine the hypertext references into expressions involving other syntax elements known to *WRspice*. The registers available through the **S** and **R** buttons to the left of the prompt line can be used to save and restore plot command strings.

The *WRspice* parser can distinguish the expressions, however in some cases the user must intervene to force an expected result. For example,

```
v(1) -v(2)
```

will be interpreted as `(v(1)-v(2))`, i.e., the difference. To force a unary minus interpretation, one can enclose the second token in double quotes or parentheses, i.e. `v(1) "-v(2)"` will plot `v(1)` and negative `v(2)`. Note that white space is not considered when interpreting the expression, and is required only to separate identifier names.

One should refer to the *WRspice* documentation for a complete description of the syntax accepted by the **plot** command. The command line can also contain keyword assignments which override defaults used when composing the plot. It is also possible to produce X-Y plots by using the "`vs`" keyword. The expression following "`vs`" will be used as the X scale for the other expressions.

The color used to render a plot reference mark in the schematic will be the same color used for the plot trace of the result to which the corresponding hypertext contributes (however, if the user has changed the plotting colors in *WRspice* or *Xic*, this may not be true). The number (or letter) enclosed by the plot mark represents a count of the hypertext entries found in the prompt line, left to right, starting with 1.

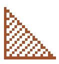
If *Xic* detects a syntax error, one or more plot marks may be rendered with "no" color (actually the highlighting color is used). This is also true for the marks used in the X-scale of an X-Y plot.

The **Enter** key terminates entry, and creates the plot if *WRspice* is available, and the circuit has been simulated with the **run** command. In the **deck** command, the string will be added to the SPICE listing, when generated, as a `.plot` line.

While the **plot** command is active, it is possible to select text labels in the normal way, other selections are inhibited. This allows labels to be selected and modified without having to explicitly terminate the **plot** command first.

The command text and mark locations are saved with the cell data when written to disk, thus the **plot** command string is persistent.

7.14 The polyg Button: Create/Edit Polygons



The **polyg** button is used to create and modify polygons. In electrical mode, this functionality is available from the **poly** menu selection brought up by the **shapes** button. A polygon is created by clicking the left mouse buton on each vertex location in sequence. The vertices can be undone and redone with the **Tab** key and **Shift-Tab** combination, which are equivalent to the **Undo** and **Redo** commands. Vertex entry is terminated, and a new polygon potentially created, by clicking on the initial point (marked with a cross), or double-clicking the last point, or by pressing the **Enter** key. At least three distinct vertices must have been defined, and the polygon must pass some "normality" tests, for successful object creation.

The PixelDelta variable can be set to alter the value, in pixels, of the snap distance to the target when clicking to terminate. By default, the snap distance is 3 pixels, so clicking within this distance of the initial point will terminate entry rather than add a new vertex.

While the command is active in physical mode, the cursor will snap to horizontal or vertical edges of existing objects in the layout if the edge is on-grid, when within two pixels. When snapped, a small dotted highlight box is displayed. This makes it much easier to create abutting objects when the grid snap spacing is very fine compared with the display scaling. This is also applied to the first vertex of polygons being created, facilitating point list termination. This feature can be controlled from the **Edge Snapping** group in the **Snapping** page of the **Grid Setup** panel.

When adding vertices during polygon creation, the angle of each segment can be constrained to a multiple of 45 degrees with the **Constrain angles to 45 degree multiples** check box in the **Editing Setup** panel from the **Edit Menu**, in conjunction with the **Shift** and **Ctrl** keys. There are three modes: call them “no45” for no constraint, “reg45” for constraint to multiples of 45 degrees with automatic generation of the segment from the end of the 45 section to the actual point, and “simp45” that does no automatic segment generation. The “reg45” algorithm adds a 45 degree segment plus possibly an additional Manhattan segment to connect the given point. The “simp45” adds only the 45 degree segment. The mode employed at a given time is given by the table below. The Constrain45 boolean variable tracks the state (set or not set) of the check box.

Constrain45 not set		
	Shift up	**Shift** pressed
Ctrl up	no45	reg45
Ctrl pressed	simp45	simp45

Constrain45 set		
	Shift up	**Shift** pressed
Ctrl up	reg45	no45
Ctrl pressed	simp45	no45

In physical mode, a new polygon is tested for reentrancy and other problems, and a warning message is issued if a pathology is detected. The new polygon is *not* removed from the database if such an error is detected. It is up to the user to make appropriate changes.

In electrical mode, if the current layer is the SCED layer, the polygon will be created using the ETC2 layer, otherwise the polygon will be created on the current layer. It is best to avoid use of the SCED layer for other than active wires, for efficiency reasons, though it is not an error. The **Change Layer** command in the **Modify Menu** can be used to change the layer of existing objects to the SCED layer, if necessary. The outline style and fill will be those of the rendering layer. Polygons have no electrical significance, but can be used for illustrative purposes.

7.14.1 Polygon Vertex Editing

On entering the **polyg** command, if a polygon is selected, a vertex editing mode is active on all selected polygons. Each vertex of the selected object is marked with a small highlighting box. Clicking on the edge of a selected polygon away from an existing vertex will create a new vertex, which can subsequently be moved.

In order to operate on a vertex, it must be selected. A vertex can be selected by clicking on it, or by dragging over it. Any number of vertices can be selected. After the selection operation, selected vertices are shown marked with a larger box, and unselected vertices are not marked. Additional vertices can be selected, and existing selected vertices unselected, by holding the **Shift** key while clicking or dragging over vertex locations. Selecting a vertex a second time will deselect it.

Selected vertices can be deleted by pressing the **Delete** key. This will succeed only if after vertex removal the object does not become degenerate. In particular, one can not delete the object in this manner.

The selected vertices can be moved by dragging or clicking twice. The selected vertices will be translated according to the button-down location and the button up location, or the next button-down location if the pointer did not move. While the translation is in progress, the new borders are ghost-drawn.

All vertex operations can be undone and redone through use of the **Undo** and **Redo** commands.

With vertices selected, pressing the **Esc** or **Backspace** keys will deselect the vertices and return to the state with all vertices marked.

While in the **polyg** command, with no object in the process of being created, it is possible to change the selected state of polygon objects, thus displaying the vertices and allowing vertex editing. Pressing the **Enter** key will cause the next button 1 operation to select (or deselect) polygon objects. This can be repeated arbitrarily. When one of these objects is selected, the vertices are marked, and vertex editing is possible.

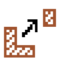
If the vertex editor is active, i.e., a selected polygon is shown with the vertices marked, clicking with the **Ctrl** button pressed will start a new polygon, overriding the vertex editor. This can be used to start a new polygon at a marked vertex location, for example. Without **Ctrl** pressed, the vertex editor would have precedence and would select the marked vertex instead of starting a new polygon.

While moving vertices, holding the **Shift** key will enable or disable constraining the translation angle to multiples of 45 degrees. If the **Constrain angles to 45 degree multiples** check box in the **Editing Setup** panel from the **Edit Menu** is checked, **Shift** will disable the constraint, otherwise the constraint will be enabled. The **Shift** key must be up when the button-down occurs which starts the translation operation, and can be pressed before the operation is completed to alter the constraint. These operations are similar to operations in the **Stretch** command.

7.14.2 Wire to Polygon Conversion

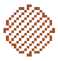
If any non-zero width wires are selected before the **polyg** command is entered, the user is given the option of changing the database representation of these objects to polygons. Is is not possible to convert polygons to wires (except via the **Undo** command if the polygon was originally a wire).

7.15 The put Button: Extract From Yank Buffer



The **put** command allows the contents of the yank buffers to be added to the current cell. This command is available in physical mode. When parts of objects are erased with the **erase** command, the erased pieces are added to a five-deep yank buffer queue. When the **put** button becomes active, the most recent deletion is ghost drawn and attached to the pointer. Clicking will place the contents of the buffer at the location of the pointer. The yank buffers can be cycled through with the arrow keys.

7.16 The round Button: Create Disk Object



The **round** button, only available in physical mode, will create a disk polygon object. The number of sides can be altered with the **sides** command. If the user presses and holds the **Shift** key after the center location is defined, and before the perimeter is defined by either lifting button 1 or pressing a second time, the current radius is held for x or y. The location of the shift press defines whether x is held (pointer closer to the center y) or y is held (pointer closer to the center x). This allows elliptical objects to be generated.

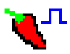
The **Ctrl** key also provides useful constraints. Pressing and holding the **Ctrl** key when defining the radius produces a radius defined by the pointer position projected on to the x or y axis (whichever is closer) defined from the center. Otherwise, off-axis snap points are allowed, which may lead to an unexpected radius on a fine grid.

When the command is expecting a mouse button press to define a radius, the value as defined by the mouse pointer (in microns) is printed in the lower left corner of the drawing window, or the X and Y values are printed if different. Pressing **Enter** will cause prompting for the value(s), in microns. If one number is given, a circular radius is accepted, however one can enter two numbers separated by space to set the X and Y radii separately.

While the command is active in physical mode, the cursor will snap to horizontal or vertical edges of existing objects in the layout if the edge is on-grid, when within two pixels. When snapped, a small dotted highlight box is displayed. This makes it much easier to create abutting objects when the grid snap spacing is very fine compared with the display scaling. This feature can be controlled from the **Edge Snapping** group in the **Snapping** page of the **Grid Setup** panel.

If the SpotSize variable is set to a positive value, or the `MfgGrid` has been given a positive value in the technology file, tiny round and donut figures are constructed somewhat differently. the figure is constructed somewhat differently. Objects created with the **round** and **donut** buttons will be constructed so that all vertices are placed at the center of a spot, and a minimum number of vertices will be used. The **sides** number is ignored. This applies only to figures with minimum radius 50 spots or smaller; the regular algorithm is used otherwise. An object with this preconditioning applied should translate exactly to the e-beam grid. See E.11 for more information.

7.17 The run Button: Run SPICE Analysis



The **run** button, available only in electrical mode, will establish interprocess communication with the *WRspice* program. If a link can not be established, the **run** command terminates with a message. If connection is established, then a SPICE run of the circuit is performed.

The user is first prompted for the *WRspice* analysis command string to run. This should be in a format understandable to *WRspice* as an interactive-mode command. During prompting, the last six unique analysis commands entered are available and can be cycled through with the up and down arrow keys.

The first word in the analysis string is checked, and only words from the following list will be accepted:

```
ac     loop   run   tran
check  noise  send
dc     op     sens
disto  pz     tf
```

The "`send`" keyword is not a *WRspice* command. If given, the circuit will be sent to *WRspice* and sourced, but no analysis is run. Other commands can be sent to *WRspice* with the **spcmd** button.

The link is established to the SPICE server (`wrspiced` daemon) named in the SPICE_HOST environment variable, or the SpiceHost "!set" variable (which overrides the environment). If neither is set, *Xic* will attempt to attach to *WRspice* on the local machine.

By default, the *WRspice* toolbar is visible when a connection has been established. This gives the user more control over *WRspice* by providing access to the visual tools. If the NoSpiceTools variable is set (with the **!set** command), the toolbar will not be visible.

During a simulation run, a small pop-up window appears, which contains a status message, and a **Pause** button. Pressing **Pause** will pause the analysis. It can be resumed by pressing the **run** button again. The analysis can also be paused by pressing **Ctrl-c** in the controlling terminal (xterm) window. A **Ctrl-c** press over a drawing window goes to *Xic*, where it stops redraws and other functions as usual.

Xic is notified when a run is paused from *WRspice* (using the red X button in the toolbar), and will change state accordingly. However, *Xic* is *not* notified when a run is restarted from *WRspice* (with the green triangle button in the toolbar), and will continue to assume that *WRspice* is inactive. In this case, commands from *Xic* that communicate with *WRspice* will pause any analysis in progress before executing. The user will have to resume the analysis manually after the operation completes, either with the **run** button or from the *WRspice* toolbar.

This affects the **plot**, **iplot**, and **run** buttons, and the **!spcmd** command. When a run is started or resumed with the **run** button in *Xic*, these features are locked out, producing a "WRspice busy" message, and the run in progress is not affected.

The node connectivity is recomputed, if necessary, before the run. If the variable CheckSolitary is set with the **!set** command, then warnings are issued if nodes with only one connection are encountered. A SPICE file is generated internally, and transmitted to *WRspice* for evaluation. Only devices and subcircuits that are "connected" will be included in the SPICE file. A device or subcircuit is connected if one of the following is true:

- There are two or more non-ground connections.
- There is one non-ground connection and one or more grounds.
- There is one non-ground connection and no opens.
- There is one non-ground connection and the object is a subcircuit.

As a final step before sending the circuit text to SPICE, *Xic* will recursively expand all `.include` and `.lib` lines, replacing the `.include` lines with the actual file text, and the `.lib` lines with the indicated text block from the library. This is to handle the case where *WRspice* is located on a remote machine, and the user's files are on the local machine. As in *WRspice*, `.inc` is a synonym for `.include`, and the '`h`' option (strip '$' comments for HSPICE compatibility) is recognized.

The `.include` and `.lib` lines are generally inserted into the SPICE text using the `spicetext` label mechanism. There may be occasions where the expansion of these lines by *Xic* is undesirable, such as when the included file resides on the SPICE host, or one wishes to use the *WRspice* `sourcepath` variable to resolve the file. To this end, the user can use the `.spinclude` keyword rather than `.include`, and

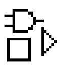
`.splib` rather than `.lib`. The `.sp` directives use the same syntax as the normal keywords, however *Xic* will not attempt to expand these directives, rather it changes the keyword to the normal directive ("`.include`" or "`.lib`"). Thus, *WRspice* will see and handle these inclusions.

WRspice release 2.2.60 and later recognize `.spinclude` as a synonym for `.include`. This allows *WRspice* to be able to directly source top-level cell files, where the SPICE listing may contain `.spinclude` lines, without syntax errors. *WRspice* release 2.2.62-2 and later recognize `.splib` as a synonym for `.lib`, and is able to handle `.lib` constructs sent from *Xic*.

Sometimes it may be desirable to place the output of a SPICE run initiated from *Xic* into a rawfile, rather than saving the output internally. To do this, use the `spicetext` labels to add an analysis string, such as "`spicetext .tran 1p 1000p`" (note that the '.' ahead of "tran" is necessary). One can also add a save command using "`spicetext *#save v(1) ...`" to save only a subset of the circuit variables. The "`*#`" means that the save is executed as a shell command, "`.save`" is ignored since *WRspice* is in interactive mode. Then, for the analysis string from *Xic*, use "`run` *filename*", where *filename* is the name for the rawfile. The run will be performed, but the output data will go to the file, so don't expect to see it with the **plot** command. If the *filename* is omitted, the run will be performed with internal storage as usual.

The **!spcmd** command can be used to give arbitrary commands to *WRspice*.

7.18 The shapes Button: Add Predefined Features



The **shapes** button appears in the electrical mode side menu. Pressing this button provides a pull-down menu of different outlines that can be applied to drawings. These outlines have no electrical significance, but can be used for illustrative purposes. In particular, in symbolic mode, this facilitates creating symbol representations. After a selection is made from the pull-down menu, the shape outline is ghost-drawn and attached to the pointer. The object is placed at locations where the user clicks.

The current choices in the pull-down menu are:

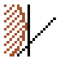
box	Create a box, like the physical mode **box** command.
poly	Create a polygon, like the physical mode **polyg** command.
arc	Create an arc, similar to the physical mode **arc** command.
dot	Place a dot (an octagonal polygon).
tri	Place a triangle (buffer symbol).
ttri	Place a truncated triangle symbol.
and	Place an AND gate symbol.
or	Place an OR gate symbol.
Sides	Set the number of sides used to approximate rounded geometry, similar to the **sides** command in physical mode.

None of these shapes have significance electrically, and for efficiency is is best to avoid using the SCED layer for these objects. In particular, arcs are actually wires, and arc vertices on the SCED layer are considered in the connectivity establishment. If the current layer is SCED when one of these objects is created, the object is instead created on the ETC2 layer. If the object must be on the SCED layer, the **Change Layer** command in the **Modify Menu** can be used to move it to that layer.

The dot, tri, ttri, and, and or choices work a little differently from the others. After selection, a ghost rendering of the shape is attached to the pointer, and the objects are placed where the user clicks. The object can be modified with the arrow keys:

Up	expand by 2
Right	expand by 10%
Down	shrink by 2
Left	shrink by 10%
Shift-Up	stretch vertically 10%
Shift-Right	stretch horizontally 10%
Shift-Down	shrink vertically 10%
Shift-Left	shrink horizontally 10%
Ctrl-Arrows	cycle through 90 degree rotations

7.19 The sides Button: Set Rounded Granularity



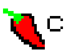
The **sides** button, available in physical mode, allows the user to set the number of sides used to approximate rounded geometries. Larger numbers give better resolution, but decrease efficiency. The number provided is the sides for a full 360 degrees, arcs will use proportionally fewer.

The setting tracks the RoundFlashSides variable. If the variable is not set, 32 sides will be used. The acceptable range is 8–256.

The setting applies when new round objects are created with the **round**, **donut**, and **arc** buttons in the physical side menu, or the equivalent script functions.

In electrical mode, the number of sides used has a separate setting using the ElecRoundFlashSides variable, which can be set from the **sides** entry in the menu presented by the **shapes** button in the electrical side menu.

7.20 The spcmd Button: Execute *WRspice* Command



This will prompt the user, in the prompt area, for a command that will be sent to *WRspice* for execution. If the user simply presses **Enter** without entering a command, or enters the command "`setup`", the **WRspice Interface Control Panel** will appear, from which the interface to *WRspice* can be set up. This panel is described in the next section.

Otherwise, a stream to *WRspice* will be established, if one is not already active, providing a means for running arbitrary *WRspice* commands. However, commands that cause *WRspice* to prompt the user for additional input (such as `setplot`) will not work properly, as the communication is one-way only and not interactive. Text output goes to the console window.

In addition to the *WRspice* commands, the client-side directive

`send` *filename*

is available. The *filename* is that of a local SPICE input file. The file will have `.include` and `.lib` lines expanded locally, and `.spinclude`, `.splib` lines will be converted to "`.include`", "`.lib`", as is done for decks created within *Xic*. The result will be sent to *WRspice* and sourced.

This operation is basically identical to the **!spcmd** command.

7.20.1 The *WRspice* Interface Control Panel

This panel appears when the user presses the **spcmd** button in the electrical side menu, and either gives no command at the prompt, or enters "`setup`". It provides entry areas for setting the variables which control the interprocess communication channel to the *WRspice* circuit simulator, and other simulation settings. Most users will probably never need to use this panel or set the associated variables as the defaults suffice in most installations.

The **WRspice Interface Control Panel** contains the following entry objects.

List all devices and subcircuits

This check box corresponds to the SpiceListAll variable. When checked, all devices and subcircuits in the schematic will be included in SPICE output. Otherwise, only devices and subcircuits that are "connected" will be included, as explained in the **deck** and **run** command descriptions.

Check and report solitary connections

This check box corresponds to the CheckSolitary variable. If checked, warning messages will be issued when electrical netlists are generated for nodes having only one connection. This affects the **run** and **deck** commands, and the **Dump Elec Netlist** command in the **Extract Menu**.

Don't show WRspice Tool Control panel

This check box corresponds to the NoSpiceTools variable. When running *WRspice* from *Xic*, by default the *WRspice* toolbar is shown, if *WRspice* is running on the local machine. This gives the user much greater flexibility and control over *WRspice*. If this check box is checked, *before* the connection to *WRspice* is established, the toolbar will not be visible.

This check box will also control toolbar visibility if the `wrspiced` daemon is used. However, this requires `wrspiced` distributed with wrspice-3.0.7 or later. **If this variable is set with an earlier `wrspiced` release, the *WRspice* connection will not work!**

Spice device prefix aliases

This group consists of a check box and a text entry area. When the box is checked, the text in the entry area will be used to set the SpiceAlias variable. This can be set to a string which will modify the printing of device names in SPICE output. The aliasing operates on the first token of device lines. The format of the string is

prefix1=*newprefix1 prefix2*=*newprefix2 ...*

This will cause lines beginning with *prefix* to have *prefix* replaced with *newprefix*. If the "=*newprefix*" is omitted, that line will not be printed. For example, to map all devices that begin with 'B' to 'J', and to suppress all 'G' devices, the string is

```
B=J G
```

Note that there can be no space around the '='. With the text entered and the box checked, the indicated mappings will be performed as SPICE text is produced.

Remote WRspice server host name

This group consists of a check box and a text entry area. When the box is checked, the SpiceHost variable is set to the text in the text area.

The text should be the name of the host which maintains a server for remote *WRspice* runs. If set, this will override the value of the SPICE_HOST environment variable. The host name specified in the SPICE_HOST environment variable and the SpiceHost **!set** variable can have a suffix "`:`*portnum*", i.e., a colon followed by a port number. The port number is the port used by the `wrspiced` program on the specified server, which defaults to 6114, the IANA registered port for this service. If the server uses a non-standard port, and the `wrspice/tcp` service has not been registered (usually in the `/etc/services` file) on this port, the port number must be provided.

Remote WRspice server host display name

This group consists of a check box and a text entry area. When the box is checked, the SpiceHost-Display variable is set to the text in the text area.

This text can be set to the X display string to use on a remote host for running *WRspice* through a `wrspiced` daemon, from *Xic* in electrical mode. This is intended to facilitate use of `ssh` X forwarding to take care of setting up permission for the remote host to draw on the local display. See the description of the piceHostDisplay variable for complete details.

Path to local WRspice executable

This group consists of a check box and a text entry area. When the box is checked, the SpiceProg variable is set to the text in the text area.

The text is the full path name of the *WRspice* executable. This is useful if there are multiple versions of *WRspice* available, or the binary has been renamed, or is not located in the standard location. If given, the value supersedes the values from environment variables or other variables (and corresponding entries) which also set a path to the SPICE executable.

Path to local directory containing WRspice executable

This group consists of a check box and a text entry area. When the box is checked, the SpiceExecDir variable is set to the text in the text area.

The text is a path to the directory to search for the *WRspice* executable. If given, the value overrides the SPICE_EXEC_DIR environment variable. The default search location is "`/usr/local/xictools/bin`", or, if the XT_PREFIX environment variable has been set, its value will replace "`/usr/local`".

Assumed WRspice program executable name

This group consists of a check box and a text entry area. When the box is checked, the SpiceEx-ecName variable is set to the text in the text area.

The text will give the expected name of the *WRspice* binary. If given, the value overrides the SPICE_EXEC_NAME environment variable. The default name is "`wrspice`".

Assumed WRspice subcircuit concatenation character

This group consists of a check box and a text entry area. When the box is checked, the Spice-SubcCatchar variable is set to the text in the text area. See the description of the variable for information about this setting.

Assumed WRspice subcircuit expansion mode

This group consists of a check box and a menu. When the box is checked, the SpiceSubcCatmode variable is set to the current menu selection. See the description of the variable for information about this setting.

7.21 The spin Button: Rotate Objects

The **spin** button, available in physical mode, allows rotation of boxes, polygons, and wires by an arbitrary angle, and subcells and labels by multiples of 45 degrees. If no objects are selected, the user is requested to select an object. With the object selected, the user is asked to click on the origin of rotation. The selected objects are ghost-drawn, and rotated about the reference point as the pointer moves.

If the **Constrain angles to 45 degree multiples** check box in the **Editing Setup** panel from the **Edit Menu** is checked, the angle will be constrained to multiples of 45 degrees. Pressing the **Shift** key will remove the constraint. If the check box is not checked, holding the **Shift** key will impose the constraint. Thus the **Shift** key inverts the effect of the check box. However, if the selected objects include a subcell or label, the angle will always be constrained to multiples of 45 degrees. The Constrain45 variable tracks the state (set or unset) of the check box.

During rotation, the angle is displayed in the lower left corner of the drawing window. The readout defaults to degrees, pressing the '`r`' key will switch to radians, and pressing the '`d`' key will switch back to degrees. Pressing the spacebar will toggle between radians and degrees.

At this point, one can click to define the rotation angle, or an absolute angle can be entered on the prompt line. To enter an angle, either press **Enter** or click on the origin marker, then respond to the prompt with an angle in degrees. In either case, the rotated boundaries of the selected objects are attached to the pointer, and new objects can be placed by clicking. Ordinarily, the original objects will be deleted, however if the **Shift** key is held while clicking, the original objects are retained. Instead of clicking, one can press the **Enter** key, which will simply rotate the selected objects around the reference point.

When the **spin** command is at the state where objects are selected, and the next button press would establish the rotation origin, if either of the **Backspace** or **Delete** keys is pressed, the command will revert the state back to selecting objects. Then, other objects can be selected or selected objects deselected, and the command is ready to go again. This can be repeated, to build up the set of selections needed.

At any time, pressing the **Deselect** button to the left of the coordinate readout will revert the command state to the level where objects may be selected to rotate.

The undo and redo operations (the **Tab** and **Shift-Tab** keypreses and **Undo/Redo** in the **Modify Menu**) will cycle the command state forward and backward when the command is active. Thus, the last command operation, such as setting the angle by clicking, can be undone and restarted, or redone if necessary. If all command operations are undone, additional undo operations will undo previous commands, as when the undo operation is performed outside of a command. The redo operation will reverse the effect, however when any new modifying operation is started, the redo list is cleared. Thus, for example, if one undoes a box creation, then starts a rotation operation, the "redo" capability of the box creation will be lost.

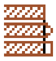
It is possible to change the layer of rotated objects. During the time that newly-rotated objects are ghost drawn and attached to the mouse pointer, if the current layer is changed, the objects that are attached can be placed on the new layer. Subcells are not affected.

How this is applied depends on the setting of the LayerChangeMode variable, or equivalently the settings of the **Layer Change Mode** pop-up from the **Set Layer Chg Mode** button in the **Modify**

Menu. The three possible modes are to ignore the layer change, to map objects on the old current layer to the new current layer, or to place all objects on the new current layer. If the current layer is set back to the previous layer before clicking to locate the new objects, no layers will change.

Note that this operation can change boxes to polygons and vice-versa. The rotation can be performed by clicking or dragging, however an angle can only be entered textually if the clicking mode is used.

7.22 The style Button: Set/Change Wire Style



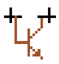
The **style** button, available in physical mode, pops up a menu of options for the presentation style of wires. The **Wire Width** choice sets the default width if no wires are selected, or changes the width of selected wires. If there are wires selected, *Xic* prompts for a new wire width for the selected wires, and the selected wires will have their widths altered. The new width should not be less than the minimum width (MinWidth design rule) for the layers.

If there are no applicable wires selected, the default wire width for the current layer is set, which is constrained to be greater or equal to the minimum width. Wires subsequently created on the present layer will have the new width.

The other choices set the default end style if no applicable wire is selected, or changes selected wires to the chosen end style if wires are selected. All selections depend on layer-specific mode. In layer-specific mode, only selected wires on the current layer are changed. Otherwise, all selected wires are changed.

The possible end styles are flush ends, extended rounded ends, and extended square ends. The extended styles project the length of the wire by half of the width beyond the terminating vertex. The button icon changes to indicate the present wire end style with a small dot.

7.23 The subct Button: Set Subcircuit Connections



The **subct** button, available in the electrical side menu, allows electrical connection terminals to be added to a circuit. The terminals are points at which electrical connections are defined, as in the SPICE subcircuit definition. Terminal definition is necessary if the circuit is to be used as a subcircuit in another circuit with connections to the instance (it is possible for a subcircuit to connect to global nets only (see 7.11), in which case the master and instances would have no terminals). The terminals are also used by the extraction system and can provide an initial association of a particular schematic net and physical conductor group.

Terminals can only be created in electrical mode. Once created, a terminal's flags may be edited so as to enable a corresponding terminal location in the physical layout. The extraction system will most often find suitable physical terminal locations automatically, however there are times when the user may need to place terminals manually, which can be done with the **Edit Terminals** button in the **Views and Operations** page of the **Extraction Setup** panel from the **Setup** button in the **Extract Menu**, while in physical mode. In electrical mode, this same button is equivalent to the **subct** button in the side menu.

Subsequent to creation with the present command, terminals can be made visible with the **terms** button in the electrical side menu. While in physical mode, the terminals will be visible in electrical windows when either the **All Terminals** or **Cell Terminals Only** check boxes in the **Show** group in the **Views and Operations** page of the **Extraction Setup** panel is checked.

The terminals must be defined in the schematic representation of the cell, whether or not the cell will ultimately be symbolic (see 7.25). The terminals can be created and deleted only in the schematic. Once created, they will be visible in the symbol view, but must be moved to the desired location by hand. In the symbol view (only) each terminal can have arbitrarily many copies or itself at different locations, each one of which is an equivalent connection point for the subcircuit. This facilitates, for example, tiling. If an equivalent connection point appears on either side of the instance, then placing a row of these instances side-by-side will automatically connect this node to all of the instances. This applies only to the symbolic representation. In the schematic, each cell terminal has a single connection point.

In *Xic*, there are two types of cell contact terminals.

Scalar terminals
: These are the "normal", single-contact terminals. These terminals actually convey the connectivity information between the parent and subcell schematics, and are the only terminals that may have corresponding terminals in the physical layout. A scalar terminal is associated with a node property, of a cell or cell instance.

Multi-contact "bus" terminals
: These terminals reference the scalar terminals and provide a means for connecting a number of these terminals to a multi-conductor net in the schematic. The use of multi-conductor nets and multi-contact terminals can greatly simplify a schematic visually. Be advised that a multi-conductor terminal only references existing scalar terminals, which must exist. These terminals are associated with a bnode property, of a cell or cell instance.

In the schematic, by default ordinary scalar terminals can only be located at connection points of the underlying geometry. These are the vertices of electrically-active wires, and device or subcell connection points. Clicking on such a point, if no terminal already exists at the point, will create a new scalar terminal at the location. The **Terminal Edit** panel will appear, which can be used to apply a name for the terminal and edit other terminal properties. The new terminal will be shown highlighted to indicate that it is the target of the **Terminal Edit** panel.

7.23.1 Virtual Terminals

If one holds the **Ctrl** key while clicking anywhere except over another terminal, a scalar terminal will be placed, whether or not it is over a circuit connection point. This is useful if the BYNAME flag is to be set for the terminal, which indicates that it will not connect by location, but by name matching only. It is also useful for implementing "virtual" terminals which connect to nothing, but satisfy connectivity references in layout vs. schematic testing, and for other purposes.

Suppose one has a subcell with physical layout only that one wishes to include in a full design hierarchy. It may not be convenient to create a schematic for the subcell, but it is desired that the connections to the subcell be included in the LVS checking of the overall design. It is possible to assign "virtual terminals" to the subcell. Virtual terminals are treated like ordinary terminals in connecting to instances of the subcell, but are ignored when creating netlists for the subcell itself.

A virtual terminal is created in the **subct** command by holding the **Ctrl** key while clicking on locations in the electrical schematic (even if the schematic is empty). They can be placed anywhere except on top of another terminal; location is not important. Once created, they can be moved or deleted like ordinary terminals.

Once placed, they will be considered in establishing the connectivity to instances of the cell, but will be ignored when establishing connections within the cell. Thus the cell looks like a "black box" with terminals. Virtual terminals can be used along with ordinary terminals if only part of the internal circuit is to be visible from the outside.

In SPICE netlists, virtual terminals will appear in the subcircuit connection list in `.subckt` and call lines, but will not be connected in the `.subckt` definition. One can use a `spicetext` label to add a `.include` line to bring in a circuit definition from a file, for example, to satisfy the references.

In the graphical display, virtual terminals of the current cell are shown with a beer-barrel outline for differentiation from the standard terminals which are square. The cell bounding box is expanded to contain all virtual terminal locations. The center of a virtual terminal is a "hot spot" for hypertext node references, i.e., clicking on the terminal center will add the associated node to the prompt line edit string in the **plot** and **iplot** commands and when creating labels or properties.

7.23.2 Multi-Contact Connectors

If the **Shift** key is held while clicking in the schematic, a new multi-contact terminal will be created. A different version of the **Terminal Edit** panel will appear, allowing the new terminal to be configured.

Multi-contact terminals reference scalar terminals, and every referenced scalar terminal should exist. The pop-up provides convenience functions for creating the "bit" terminals. In some cases, these will be made invisible and not shown in either the schematic or symbol, yet they must exist as they provide a crucial data structure required for actual connectivity.

Named and unnamed multi-conductor terminals identify their constituent bits quite differently. If a terminal is named, the name is a net expression (see 4.2.8) that unambiguously specifies the names of the scalar terminals. These terminals are referenced by name, so ordering is unimportant.

If a multi-conductor terminal is unnamed, it will at least have a default range of `[0:0]`. The terminal also has an index number that defaults to 0. The bits are the scalar terminals with indices starting with the multi-conductor terminal index value, through the width of the multi-conductor range, contiguously and increasing. In this case, terminal ordering is obviously quite important.

See the **Terminal Edit** panel description in 7.24 for a complete discussion of the conrfiguration options for multi-contact terminals (and scalar terminals, too).

7.23.3 Terminal Ordering

By default, a newly-created scalar terminal will be given the largest index number, meaning that it will be the last terminal listed when the subcircuit is represented in SPICE or other netlisting output. However, it is possible to insert new terminals at any point in the sequence.

If the user types a number while the command is active, the number will appear in the keypress buffer area for the drawing window that has the keyboard focus. If this number is within the range of existing terminal indices, then new terminals created from this window will be given this index, and existing terminals with this index or larger will have their indices incremented.

Suppose for example that the cell contains five terminals, and one needs to add a sixth, and further the new terminal should be the fourth terminal in the sequence (index number 3). While in the **subct** command, one can type “3” and note that “3” appears in the keypress buffer area. One can now click on a circuit location to create the new terminal, and note that the new terminal is given index 3, the previous 3 is now 4, etc. The backspace key can be used to clear the keypress buffer, or the next new terminal added will also be inserted as number 3. Note that one can type “0” and leave this in place, so that all new terminals will be added to the front of the list rather than the back.

The indexing and order can also be changed with the **Terminal Edit** panel.

For multi-contact terminals, the index parameter provides ordering information. The terminal order assumed by *Xic* is that a multi-contact terminal is ordered by its index, ahead of a scalar terminal with the same index. If the multi-contact terminal is named, then the index number is arbitrary, however by convention *Xic* will set the index to the index of the first (leftmost) bit. If the terminal is unnamed, the index is also the index of the first bit, and in fact this identifies the first bit.

7.23.4 Terminal Naming and Editing

If no name is given to a scalar terminal, *Xic* will use a default name, which is an underscore followed by the internal index (the number shown in the marker). Otherwise, a short descriptive name can be entered. The name must follow the rules for a scalar net expression (see 4.2.8), that is, it must be a simple text name, with or without a single index subscript. A non-default name will be displayed next to the terminal marker (the default name is assumed if the entry is an underscore followed by one or two digits).

Clicking on an existing terminal will select it, and begin a move operation. A box will be ghost-drawn and attached to the mouse pointer. If the terminal is scaler, it can be moved to a new location by clicking on a connection point not occupied by another terminal. It can be moved to a non-contact point by holding **Ctrl** while clicking, and the terminal becomes “virtual”. Multi-contact terminals can be moved to any location not already occupied by a terminal.

While a terminal is selected, pressing the **Delete** key will delete the terminal. Pressing **Backspace** or **Esc** will deselect the terminal, aborting the move operation.

If an existing terminal is clicked on with the **Shift** key held down, or double-clicked on (including being “moved” to the same location), the **Terminal Edit** panel will appear, allowing the user to edit the parameters for the terminal.

From the **Terminal Edit** panel, it is possible to make the terminal invisible. This can be applied to terminals that do not participate in the visual connections, so clutter the display needlessly. The **PageUp** and **PageDown** toggle the display of (otherwise) invisible terminals while the **subct** command is active. Invisible terminals can also be selected for editing with the **Next** and **Prev** buttons in the panel, which cycle through the terminals to edit by the index value.

In symbolic mode, terminals can not be added or deleted, however they can be moved to new locations consistent with the symbolic representation. Terminals can be moved by dragging, or by clicking on a terminal then clicking on the new location. Terminals can be placed anywhere in the symbolic representation. Further, if the **Shift** key is held during the terminal placement, the original terminal mark is retained, i.e., a copy is made. Any number of copies can be placed. Copies can be deleted by clicking to select, then pressing the **Delete** key. The last remaining instance of a terminal can not be deleted in this way, one must go to the schematic to delete the terminal.

7.24 The Terminal Edit Pop-Up: Editing Terminals

The **Terminal Edit** pop-up appears when using the **subct** button in the electrical side menu. It also appears while in physical mode and using the **Edit Terminals** button from the **Setup** page of the **Extraction Setup** panel, which is brought up with the **Setup** button in the **Extract Menu**. In either case, it provides a means for editing various properties of a terminal, including its name.

When the panel is visible, one of the terminals in the display is highlighted, and the controls in the panel represent the state for the highlighted terminal. This is the "target terminal" which will be modified by the panel.

The panel configures itself for either scalar or multi-contact terminals in electrical mode, depending on the target terminal. In physical mode, only scalar terminals exist and not all parameters are editable, and the panel configures itself accordingly. The panel will appear quite different in these three cases.

The target terminal can be changed by **Shift**-clicking or double-clicking over a different terminal. It can also be changed with the **Prev**, **Next**, and **To Index** buttons found in the panel.

Every scalar terminal has a unique index number. This is the number that is shown in the box which represents the terminal in the schematic. This represents the order of the terminals in calls to instances of the current cell. Bus terminals have an index number as well, which must be one of the scalar terminal indices. The ordering of the multi-contact terminal is at the index, but *before* the scalar terminal with the same index.

The **Prev** button will cycle the target terminal to the one with index value one less than the current index, wrapping at zero. The **Next** button will cycle the target terminal in the opposite direction. The **To Index** button and numeric entry area can be used to change the target terminal to one with the specified index, of the same type (scalar or multi-contact terminal) as the present terminal.

No actual change is made unless or until the **Apply** button is pressed. Pressing **Apply** will update the target terminal according to the entries in the panel. Changes made can be undone and redone with the standard *Xic* undo/redo operations.

Pressing the **Dismiss** button will retire the panel.

7.24.1 Electrical Scalar Terminal Editing

At the top of the panel is a **Terminal Index** numeric entry area. This can be used to change the terminals index number, and therefor order in subcircuit references. The renumbering is a two step process:

1. The present terminal is removed, and the remaining terminals are renumbered, using unique and contiguous new index values (zero based).
2. The terminal is reinserted at the given index. The terminal that had that index and those larger will have their index values incremented.

Changing the index of a scalar terminal does **not** update the multi-contact terminals! The index values used in the bus terminals may require compensating changes.

Just below is the **Terminal Name** text entry area. This will contain the name of the terminal, which can be edited by the user. The entry can be empty, in which case *Xic* will generate a name.

The **Has physical terminal** check box should be checked if the terminal will have a corresponding contact point in the physical layout. Setting this check box will allocate the internal data structure needed to maintain the association. In most cases, this will be required. It is not required if, for example, the user at this point is only concerned with a schematic for simulations. The terminal can be edited and this box checked at a later time, when the user is ready to add a layout. The box is never checked for terminals used in the schematic for special purposes that are perhaps related to simulation, that have no "real" implementation in the layout.

When the **Has physical terminal** check box is checked, the **Physical** group is un-grayed. There are two controls in this group.

Layer Binding
The **Layer Binding** menu provides a layer name that is a hint used by the extraction system when placing the physical terminal in the layout. This is set by *Xic* after extraction, and if correct should not be changed. It is set by the user when a terminal is manually placed, to resolve ambiguity about which layer the terminal connects to.

Location locked by user placement
When a terminal is manually placed, the **Location locked by user placement** check box will become checked. This indicates that the FIXED flag is set in the terminal. Terminals with this flag set will never be moved by *Xic* during extraction/association.

The location and layer must be correct or association will fail. Although *Xic* will automatically place terminals, at times this will fail and the user will have to place some terminals manually to obtain correct or complete association.

Below the **Physical** group are check boxes for setting some binary options.

Set contact by name only
This check box, when checked, sets the BYNAME flag in the terminal which changes its interpretation in the schematic (it has no effect in physical mode). Ordinarily, a terminal is placed on a "connection point" of a wire net in the schematic (i.e., a vertex), or a device or subcircuit contact point. Association of the terminal to that wire net is by location. If there is no underlying connection point, and the terminal has an assigned name, *Xic* will then attempt to add the terminal to an existing net with a matching name. If this flag is set, then the initial attempt to connect the terminal by location will be skipped. This is useful if the terminal is to be made invisible, to avoid accidental connections. The scalar wire nets can be named with the **Node (Net) Name Mapping** panel from the side menu (see 7.11).

Set terminal invisible in schematic
This check box, when checked, sets the SCINVIS flag in the terminal which prevents the terminal from being displayed in schematics. This is for terminals that are used only as bit connections for a multi-contact connector. Recall that every bit in a multi-contact connector is a scalar connector, that must exist if a connection is to be established. If connectivity is to be provided only via the multi-contact connector, the individual bits are visually superfluous and clutter the display. However, they can be made invisible in the schematic with this flag. They should probably also have the BYNAME flag set as well, so that they don't make an unintended connection by location. The setting has no effect in physical mode.

Set terminal invisible in symbol
This check box controls the analogous SYINVIS flag, which when set causes the terminal to be invisible in the symbolic representation, if any. This flag will almost always track the state of

the SCINVIS flag, but this is not an absolute requirement. It is possible for a schematic to use individual bits for connections, whereas the symbol uses a multi-contact terminal, or vice-versa.

7.24.2 Physical Terminal Editing

In physical mode, the panel allows changes only within the **Physical** group described above. That is, the **Layer Binding** choice and the **Location locked by user placement** check box are the only editable entries. These have the purpose and functionality as described above. One must return to electrical mode to change other parameters.

7.24.3 Multi-Contact Connector Editing

When the target terminal is a multi-contact connector, the panel reconfigures itself to provide the appropriate entry areas.

At the top of the panel is a numeric **Term Index** entry area. Just below this are two text entry areas with labels **Terminal Name** and **Net Expression**. A "bundle" terminal may have a separate simple text name, as well as its net expression. If given, the simple text name will be used as a name for the terminal in instance placements of the cell. The terminal in the instance will look like a pure vector terminal with the given name, and a range starting with zero and extending to the width of the bundle minus one.

If the terminal does not represent a bundle, then internally there is only one name, which is the net expression. This is obtained from the two entry areas, which should not conflict or an error will result. Probably the best approach is to use the **Net Expression** entry for the complete expression, and leave the **Terminal Name** entry blank. Alternatively, one could put a text name in the name entry, and the subscripting, without a name or with the same name, in the expression entry.

It is legitimate to not provide a name, but to provide subscripting only. In this case:

1. The subscripting is ignored, except to determine the implied width (number of conductors).

2. The connector maps the scalar terminal with index value equal to the **Term Index** entry and terminals with successive indices, the total number of which will be equal to the connector width. Thus, scalar terminal order and the **Term Index** value are critical in this case. It is up to the user to maintain consistency while editing, as indices may change. Probably, though, there is no reason to use this approach, and not supply a terminal name.

If the terminal has a name, or has a bundle net expression, then the name of every scalar terminal "bit" is well defined. These are found by name, so there is no order requirement, only an existence requirement. Furthermore, the **Term Index** entry has much less significance. It is only used to assign an order for the terminal relative to other terminals. Specifically, the terminal order is just ahead of the scalar terminal with the same index (multi-conductor terminal index values are required to be unique). *Xic* will initially assign the index as the index of the first scalar terminal referenced. This can be changed if necessary.

Below the three entry areas is a **Delete** button, which will delete the terminal if pressed. This, and all other operations, can be undone/redone with the standard *Xic* **Tab**/**Shift-Tab** keys and equivalent operations in the **Modify Menu**.

There are two check boxes for terminal visibility in the schematic and symbol, as we saw for scalar terminals. It is unlikely that the user would go to the trouble of implementing a multi-contact terminal and not have it visible, but it is possible.

The **Bus Term Bits** group provides some specialized functions for working with the scalar terminals referenced. These can be applied only if the terminal has a name or is a bundle terminal.

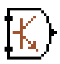
Check/Create Bits
: This will create, at the end of the scalar terminal list, any scalar terminal referenced by the present terminal and not found. Newly created scalar terminals whill have BYNAME, SCINVIS, and SYINVIS set, meaning that the terminals will be invisible and make contact by name only. The new terminals are placed at the same location as the present terminal. As they are invisible and they do not connect by location, there is no problem with this. In one way or another, the scalar terminals referenced by a multi-conductor terminal must exist for connectivity to be established, even if they are invisible and never dealt with again after creation. The **Check/Create Bits** button makes the scalar terminal creation quick and easy. Be aware, though, that it will probably still be necessary to edit these terminals to set the physical data.

Reorder to Index
: This will create missing scalar terminals as above, but in addition it will reorder the scalar terminals list so that the index values of the referenced terminals are contiguous and start with the **Term Index** value. All other considerations aside, this may be a "nice" way to organize the terminals. It is also potentially more efficient. If the net expression does not duplicate any connection bits, an internal mapping step can be skipped as it becomes an identity, saving a little memory and time. This is the same ordering used with "unnamed" terminals.

The four buttons below allow setting of the visibility flags of all of the referenced scalar terminals. It is unlikely that the flag states would vary between the bits.

The remaining buttons operate as described for scalar terminal editing.

7.25 The symbl Button: Symbolic Representation



The **symbl** button, available in electrical mode, allows instances of a cell to be shown as a symbol, rather than as a schematic. In the symbolic representation, the substructure of the cell is never shown, instead a simple figure representing the cell is displayed. This can simplify complex schematics.

When this button is active, the current cell is in symbolic mode. It is not possible to add subcircuits or devices in this mode, but any geometry added will show as the symbolic representation. If the cell is saved with this button active, then the cell and its instances will use the symbolic representation.

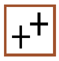
However, it is possible to apply a property to individual instances of the cell to force the display of that instance non-symbolically (as a schematic). This property can be applied with the **Property Editor**.

If the **No Top Symbolic** button in the **Main Window** sub-menu of the **Attributes Menu**, or in the sub-window **Attributes** menu, is set, the top cell will always display as a schematic in the window, whether or not the **symbl** button is pressed.

When a new cell is opened for editing, the **symbl** button will become active and the symbolic representation shown if the cell was previously saved in symbolic mode. Pressing the button a second time will revert to normal presentation.

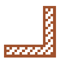
While in symbolic mode, subcircuit terminals can not be added, however existing terminals can be moved to new locations by dragging. One should first place the terminals, with the **subct** command, in normal mode. After switching to symbolic mode, the terminals can be moved to new locations, in the generally more compact symbolic representation. The actual locations of subcircuit connections is dependent upon the mode.

7.26 The terms Button: Show Subcircuit Connections



When the **terms** button is active, the electrical connection points of the subcircuits are shown. These points are placed with the **subct** command. The **terms** button is available in electrical mode only. When active, the physical terminals will be shown in physical mode windows, as if the **All Terminals** check box in the **Setup** page of the **Extraction Setup** panel was checked. This panel is obtained from the **Setup** button in the **Extract Menu**. Similarly, in physical mode, when physical terminals are visible, electrical terminals will also be visible in electrical windows, as if the **terms** button was active.

7.27 The wire Button: Create/Edit Wires



The **wire** button is used to create or modify wires. A wire is created by clicking the left mouse button on each vertex location in sequence. The vertices can be undone and redone with the **Tab** key and **Shift-Tab** combination, which are equivalent to the **Undo** and **Redo** commands. Vertex entry is terminated, and a new wire created, by clicking a second time on the last point, or by pressing the **Enter** key.

The PixelDelta variable can be set to alter the value, in pixels, of the snap distance to the target when clicking to terminate. By default, the snap distance is 3 pixels, so clicking within this distance of the last point will terminate entry rather than add a new vertex.

In electrical mode, wires are used to connect devices into circuits. Vertices are recognized as connecting points, and are created where the wire crosses a device or subcircuit terminal or a vertex of another wire. The **Connection Dots** button in the **Attributes Menu** can be used to display connections. The vertices can be edited to remove or reestablish connections.

In electrical mode, entering the **wire** command will switch the current layer to the SCED (active) layer. The current layer can be changed if necessary, but without the reverting it was too easy to create wires on another layer, sometimes difficult to visually differentiate, that will not be electrically active in the schematic causing the circuit to not work.

While the command is active in physical mode, the cursor will snap to horizontal or vertical edges of existing objects in the layout if the edge is on-grid, when within two pixels. When snapped, a small dotted highlight box is displayed. This makes it much easier to create abutting objects when the grid snap spacing is very fine compared with the display scaling. This is also applied to the last vertex of wires being created, facilitating point list termination. This feature can be controlled from the **Edge Snapping** group in the **Snapping** page of the **Grid Setup** panel.

When adding vertices during wire creation, the angle of each segment can be constrained to a multiple of 45 degrees with the **Constrain angles to 45 degree multiples** check box in the **Editing Setup** panel from the **Edit Menu**, in conjunction with the **Shift** and **Ctrl** keys. There are three modes: call them “no45” for no constraint, “reg45” for constraint to multiples of 45 degrees with automatic generation of the segment from the end of the 45 section to the actual point, and “simp45” that does no automatic segment generation. The “reg45” algorithm adds a 45 degree segment plus possibly an additional Manhattan segment to connect the given point. The “simp45” adds only the 45 degree segment. The mode employed at a given time is given by the table below. The Constrain45 variable tracks the state (set or not set) of the check box.

Constrain45 not set		
	Shift up	**Shift** pressed
Ctrl up	no45	reg45
Ctrl pressed	simp45	simp45

Constrain45 set		
	Shift up	**Shift** pressed
Ctrl up	reg45	no45
Ctrl pressed	simp45	no45

In physical mode, three end styles are available for nonzero width wires: Flush, Rounded, and Extended. The end style and the default width are set from the menu provided by the **style** button. The end style of selected wires can be changed from this menu, from within the **wire** command or without.

The width of wires on a particular layer, or the widths of existing wires, can be set of changed with the **Wire Width** button in the menu brought up with the **style** button. Zero-width wires are accepted into the database if they contain more than one point. In physical mode, they probably should not be used, and they will, of course, fail DRC tests. They are allowed in the off chance that the user uses them for annotation purposes. Such lines will be invisible, however, unless the layer pattern is outlined or solid. In electrical cells, zero-width wires are commonly used for the connecting lines, and there is no question of their legality in electrical cells. The width of selected wires can be changed with this menu command, from within the **wire** command or without.

If the first vertex of a wire being created falls on an end vertex of an existing wire on the same layer, the new wire will use the same width and end style as the existing wire, overriding the defaults. The completed new wire will be merged with the existing wire, unless merging is disabled. Merging can be controlled from the **Editing Setup** panel from the **Edit Menu**, and note also that the NoMerge layer attribute will prevent merging.

Wires with a single vertex are acceptable if the width is nonzero and the end style is rounded or extended. These are rendered as an octagon or box, respectively, centered on the vertex.

Existing wires can be converted to polygons through selection and execution of the **polyg** command.

7.27.1 Wire Vertex Editor

On entering the **wire** command, if a wire is selected, a vertex editing mode is active on all selected wires. Each vertex of the selected object is marked with a small highlighting box. Clicking on a selected wire away from an existing vertex will create a new vertex, which can subsequently be moved.

In order to operate on a vertex, it must be selected. A vertex can be selected by clicking on it, or by dragging over it. Any number of vertices can be selected. After the selection operation, selected vertices are shown marked with a larger box, and unselected vertices are not marked. Additional vertices can be selected, and existing selected vertices unselected, by holding the **Shift** key while clicking or dragging over vertex locations. Selecting a vertex a second time will deselect it.

Selected vertices can be deleted by pressing the **Delete** key. This will succeed only if after vertex removal the object does not become degenerate. In particular, one can not delete the object in this manner.

The selected vertices can be moved by dragging or clicking twice. The selected vertices will be translated according to the button-down location and the button up location, or the next button-down location if the pointer did not move. While the translation is in progress, the new borders are ghost-drawn.

All vertex operations can be undone and redone through use of the **Undo** and **Redo** commands.

With vertices selected, pressing the **Esc** or **Backspace** keys will deselect the vertices and return to the state with all vertices marked.

While in the **wire** command, with no object in the process of being created, it is possible to change the selected state of wire objects, thus displaying the vertices and allowing vertex editing. Pressing the **Enter** key will cause the next button 1 operation to select (or deselect) wire objects. This can be repeated arbitrarily. When one of these objects is selected, the vertices are marked, and vertex editing is possible.

If the vertex editor is active, i.e., a selected wire is shown with the vertices marked, clicking with the **Ctrl** button pressed will start a new wire, overriding the vertex editor. This can be used to start a new wire at a marked vertex location, for example. Without **Ctrl** pressed, the vertex editor would have precedence and would select the marked vertex instead of starting a new wire.

While moving vertices, holding the **Shift** key will enable or disable constraining the translation angle to multiples of 45 degrees. If the **Constrain angles to 45 degree multiples** check box in the **Editing Setup** panel from the **Edit Menu** is checked, **Shift** will disable the constraint, otherwise the constraint will be enabled. The **Shift** key must be up when the button-down occurs which starts the translation operation, and can be pressed before the operation is completed to alter the constraint. These operations are similar to operations in the **Stretch** command.

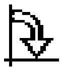
7.27.2 Associated Net Name Label

In electrical mode, wires that participate in schematic connectivity can have an associated text label. The text provides a name for the net (node) that contains the wire, and is equivalent to the placement of a named terminal device (see 7.5.1) at a vertex of the wire.

To create and bind a label to a wire, first select the wire. Then, press the **label** button in the side menu. Enter the text, and place the label in the normal way. The text in the label will be taken as a candidate net name (see 7.11) for the net containing the wire.

Unlike unlabeled wires, a wire with a label will never be merged with adjacent wires. Labeled wires play an important role in the connectivity of some schematics, by defining multi-conductor wire nets, and providing the “taps” to access the net. Complete information is provided in the Connectivity Overview in 4.2.7 and the sections that follow.

7.28 The xform Button: Current Transform Panel



The **xform** button in the side menu brings up the **Current Transform** panel, which allows the current transform to be set. The current transform is applied to newly-placed subcells, and to objects which are moved or copied.

The transform that is applied to an instance of a cell is saved in an irreducible form in the database representation of the instance. The irreducible form is an optional reflect-y ($y \rightarrow -y$), followed by an optional rotation, followed by the translation. This maps directly to the format used in GDSII files. However, the "current transform" applies rotation *before* the reflection, so that on screen, a reflect-x, for example, will flip the object's x coordinates independent of any rotation angle, which is what users tend to expect. The transform string printed on unexpanded instances and on the status line reflects this, i.e., forms like "`R45M`" imply a 45 degree rotation followed by a reflect-y ("`M`" always denotes reflect-y, reflect-x is equivalent to some other rotation and reflect-y combination). However, the transformation shown in an **Info** window will be reflect-y followed by a 315 degree rotation (in this example), since the internal representation performs the reflection before the rotation.

If the current transform is set to something other than the default identity transform, the transform code is printed on the status line.

The following buttons and input fields are available in the **Current Transform** panel.

Angle

This choice menu allows setting the rotation component of the current transform. The menu allows a choice of rotations in increments of 90 degrees in electrical mode or 45 degrees in physical mode.

Pressing and holding the **Ctrl** key and then pressing the left or right arrow keys cycles through the transformation angles, whether or not the **Current Transform** panel is visible. The right arrow increases the angle, the left arrow decreases the angle.

Reflect X

Add a reflection of the x-axis to the current transform. The X-reflection is toggled by the **Ctrl-Down Arrow** key sequence, whether or not the **Current Transform** panel is visible.

Reflect Y

Add a reflection of the y-axis to the current transform. The Y-reflection is toggled by the **Ctrl-Up Arrow** key sequence, whether or not the **Current Transform** panel is visible.

Magnification

This entry field allows setting of the magnification component of the current transform. Any number from 0.001 through 1000.0 can be entered.

Identity Transform

This button will save the current parameters to internal storage, and reset these values to the default state (no transformation). The saved state can be restored with the **Last Transform** button.

When the panel first appears, this button will have the keyboard focus if the current transform is not the identity transform. The user can press **Enter** to "press" the button. This will cause the focus to switch to the **Dismiss** button, so that another **Enter** press will retire the panel. Thus, one can very quickly restore the identity transform using the **xform** button accelerator ("`xf`") followed by pressing the **Enter** key twice.

Last Transform

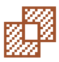
This button will restore the state of the current transform last saved with the **Identity Transform** button, or one of the recall buttons. If no state has been saved, the identity transform (the default) is set. Note that there is separate storage for the current transform in electrical and physical modes.

When the panel first appears, if the current transform is the identity transform, this button will have the keyboard focus. In this case, the same key sequence as described above can be used to quickly restore the last transform.

Store and Recall

There are five internal registers for storage of transformation parameters. Separate registers are used in electrical and physical modes. Pressing these buttons will either save the current parameters to a register, or set the parameters from a register. After a recall, the original parameters can be restored with the **Last Transform** button.

7.29 The xor Button: Exclusive-OR Objects



The **xor** button facilitates inverting the polarity of layers, and is available only in physical mode. The operation is similar to the **box** command, however all previously existing boxes, polygons, and wires on the same layer which overlap the created box become holes in the new box. Only boxes, polygons, and wires are inverted, other structures are covered. When a wire is partially xor'ed, the part of the wire outside of the xor region becomes a polygon. The **!layer** command can also be used to invert layer polarity, and is recommended when an entire cell is to be inverted.

While the command is active in physical mode, the cursor will snap to horizontal or vertical edges of existing objects in the layout if the edge is on-grid, when within two pixels. When snapped, a small dotted highlight box is displayed. This makes it much easier to create abutting objects when the grid snap spacing is very fine compared with the display scaling. This feature can be controlled from the **Edge Snapping** group in the **Snapping** page of the **Grid Setup** panel.

The **box**, **erase**, and **xor** commands participate in a protocol that is handy on occasion.

Suppose that you want to erase an area, and you have zoomed in and clicked to define the anchor, then zoomed out or panned and clicked to finish the operation. Oops, the **box** command was active, not **erase**. One can press **Tab** to undo the unwanted new box, then press the **erase** button, and the **erase** command will have the same anchor point and will be showing the ghost box, so clicking once will finish the erase operation.

The anchor point is remembered, when switching directly between these three commands, and the command being exited is in the state where the anchor point is defined, and the ghost box is being displayed. One needs to press the command button in the side menu to switch commands. If **Esc** is pressed, or a non-participating command is entered, the anchor point will be lost.

Chapter 8

The File Menu: *Xic* Input/Output

The **File Menu** contains commands for opening, listing, and saving files and cells. The printer interface for hard-copy plots is also found in this menu.

Some of the menu commands bring up more complicated panels which themselves may contain various command buttons and other objects. Most of these windows can be moved by pressing the left mouse button in the area outside of any buttons, or on a label object, and dragging the outline to the desired location. This applies to the error message and information windows that pop up under certain circumstances. These windows can also be deleted by double clicking with button 2 in the area outside of buttons or other objects.

The table below lists the commands found in the **File Menu**, along with the internal command name and function. The **OpenAccess Libs** button will appear only if the OpenAccess plug-in is loaded.

File Menu			
Label	**Name**	**Pop-up**	**Function**
File Select	`fsel`	**File Selection**	Open file
Open	`open`	none	Open new cell or file
Save	`sv`	**Modified Cells**	Save modified cells
Save As	`save`	none	Save file, rename
Save As Device	`sadev`	**Device Parameters**	Electrical mode only, apply defaults and save device
Print	`hcopy`	**Print Control Panel**	Hard copy plot
Files List	`files`	**Path Files Listing**	List search path files
Hierarchy Digests	`hier`	**Cell Hierarchy Digests**	List of Cell Hierarchy Digests
Geometry Digests	`geom`	**Cell Geometry Digests**	List of Cell Geometry Digests
Libraries List	`libs`	**Libraries**	List libraries
OpenAccess Libs	`oalib`	**OpenAccess Libraries**	List OA libraries (with OA only)
Quit	`quit`	none	Exit *Xic*

8.1 The File Select Button: Pop Up File Selection Panel

The **File Select** button in the **File Menu** brings up the **File Selection** panel. The **File Selection** panel can be used to select files to edit, or to manage files and directories on disk. The button can be used to bring up more than one **File Selection** panel, and drag/drop can be used to move files and directories. From this button, the **File Selection** panel will list files in the current directory, but this can be changed from the panel.

8.2 The Open Button: Open Cell or File

The **Open** button in the **File Menu** is used to read a file and/or load a cell for editing. The button presents a drop-down menu containing the names of the last eight cells opened for editing, plus "**new**" and "**prev**" entries. If **prev** is clicked, the last opened cell (if any) will be reopened in the main window.

If one holds down **Shift** while selecting one of cells from the history list, the **Cell Placement Control** panel will appear with that cell added as the current master. This applies to cell names and not **new** or **prev**. This is a quick backdoor for instantiating cells recently edited.

Selecting **new** with the **Shift** button held down opens a new cell with a unique name. This can be used for experimentation, or for other purposes. The **Save As** command can be used to save the contents to a cell with a more descriptive name, if desired.

Otherwise selecting **new** will use the prompt line to request a file or cell name to open. The internal keyword `open` is associated with this button. The accelerator actually maps to the **new** button in the pop-up menu, i.e., the accelerator will cause prompting for the name of a file or cell to open.

The default name used in the prompt of the cell to edit will be one of the following. Each of these sources is tested in order, and the first one that is visible and has a selection will yield the default name.

- A selection in the **File Selection** pop-up from the **File Select** button in the **File Menu**.
- A selection in the **Cells Listing** pop-up from the **Cells List** button in the **Cell Menu**.
- A selection in the **Files Listing** pop-up from the **Files List** button in the **File Menu**, or its **Content List**.
- A selection in the **Content List** of the **Libraries** pop-up from the **Libraries List** button in the **File Menu**.
- A selection in the **Cell Hierarchy Tree** pop-up from the **Show Tree** button in the **Cell Menu** or from the **Tree** button in the **Cells Listing** pop-up.
- A cell name that is selected in the **Info** pop-up, from the **Info** button in either the **View Menu** or the **Cells Listing** pop-up.
- The name of a selected subcell in the drawing window, the most recently selected if there is more than one.
- The next cell from the command line invoking *Xic*.
- The current cell name.

8.2.1 Input to the Open Command

The text given to the **Open** command must contain at least one and at most two names. If a name contains white space, the name must be quoted with double quote marks (`"name with space"`) for it to be recognized as a single token. The first name is generally that of a multi-cell source, such as a path to a layout file. The second name, which is optional, is the name of a cell from that source to open as the current cell. If not given, depending on the source, either a default cell is opened, or the user is presented a list of cells from which to choose. If a single name is given, it can also be the name of a cell in memory, or the name of a cell resolvable through a library or the search path for native cells.

In short, the first or only name given can be one of the following.

- The name of an OpenAccess library, if the OpenAccess plug-in has been loaded.
- A path to a layout file in a supported format.
- The access name of a Cell Hierarchy Digest (CHD) in memory.
- A path to a CHD file on disk.
- A URL to a layout file on a remote server. This can also apply to a CHD file, but the layout file referenced by the CHD must be available locally.
- The name of a library file.

In each of the cases above, a second name can appear, giving the name of a cell to open. If no cell name is given, the action depends on the type of source. An OpenAccess library source requires that a cell name be given, otherwise the OpenAccess database is not consulted.

If no cell name is given and the source is a layout file containing only one top-level cell, that cell will be opened. If there are multiple top-level cells, a pop-up will appear allowing the user to choose which cell to open. These calls will already be in memory, the choice simply defines the current cell for editing.

If the source is a CHD and no cell name is given, the CHD's default cell will be opened. This is either a cell configured into the CHD, or the first (lowest offset) top-level cell found in the original layout file. There will never be a selection pop-up with a CHD source.

If the file is a library file, the second argument should be one of the reference names from the library, or the name of a cell defined in the library. If no second name is given, a pop-up listing the library contents will appear, allowing the user to select a reference or cell.

The **Open** command can access the internet. The name given to the **Open** command can be in the form of a URL, followed by options. The URL must begin with "`http://`" or "`ftp://`", and the file is expected to be suitable *Xic* input.

There is presently only one option that can follow the url:

`-o` *filename*
Ordinarily a temporary file is used for downloading, which is destroyed. The user must save the hierarchy to retain a copy on the user's machine. If this option is given, the downloaded file will be saved in the given file and not destroyed.

If the name can not be resolved as a source archive as described above, it may be the name of one of the special library files. If not, it is taken as a name for a cell. If it can not be resolved as a known cell, a new, empty cell is created with that name.

- The name of the model or device library file.
- The name of a cell already in memory.
- The name of a cell resolvable through open libraries or the native cell search path.

- The name of a new cell to create and open.

If the name of the file given is that of the present model library (default "`model.lib`") or device library (default "`device.lib`"), the library file is first copied into the current directory if it doesn't exist there, and the file in the current directory is then opened for text editing. These files contain the devices and some of the models used in electrical mode for producing SPICE files.

Cells can also be opened for editing within *Xic* by dragging the name from a file manager and dropping in the main drawing window, or by pressing the **Ok** or **Open** buttons in the **File Selection** panel. Files can also be opened from the **Open** buttons in the files, cells, and contents listing pop-ups in the **File Menu**. These are all equivalent to opening the cell with the **Open** command, so that the information in this section applies in those cases.

If the name string given to edit matches the name of a cell in memory, the editing context is switched to that cell, and no disk file is read in this case. However, if the name given to edit contains a directory separation character, i.e., is a path, then *Xic* will always attempt to read the file from disk. Thus, if the user wants to re-read a native cell file from disk, if the cell is already in memory, the user should add a path prefix to the name. For example "`./noname`", assuming `noname` is in the current directory, will force *Xic* to read the disk file, even if the `noname` cell is already in memory.

The interpretation of any path prefix which is included with the name of a native file to open for editing is set by the variables NoReadExclusive and AddToBack. The top level cell will always be read from the given file if a path to the file is specified. Subcells are resolved by cell name only through the search path. The search path is modified during the read according to the state of the NoReadExclusive and AddToBack variables.

All of the settings in the **Setup** page of the **Import Control** panel (from the **Convert Menu**) apply. However, none of the options, such as layer filtering or cell name modification, found in the **Read File** page of the same panel apply in this case. If these options are needed, the **Read File** button in this page should be used to read the file, rather than the **Open** command. Note that this is different from pre-3.0.0 releases, in which cell name case changes and file-based aliasing were supported in the **Open** command.

The table in 14.1 lists the variables and modes that apply to the **Open** and similar commands.

8.2.2 Reading Input With the Open Command

While a layout file is being read and processed, a log file is written. This file contains a record of messages emitted during the conversion. If during a conversion an error or warning message is emitted, a file browsing window containing the log file will appear when the conversion is complete, though this can be suppressed by setting the NoPopUpLog variable. These messages also appear on the prompt line during the conversion. The file browser is a read-only version of the text editor window (see 3.13.2). The log files can be accessed from the **Log Files** button in the **Help Menu**.

When reading a layout file, there is a message updated periodically on the prompt line indicating bytes read. One can abort the read with **Ctrl-c**, and a 'y' response to the resulting prompt. It is advisable to clear the cells from the partially read hierarchy from memory with the **Clear** button in the **Cells Listing** pop-up.

CGX and GDSII files that have been compressed with the GNU gzip program or have been written in compressed form by *Xic* can be read in directly, whether or not the file name contains the standard "`.gz`" suffix. Support for compressed files extends to CGX and GDSII only (OASIS files use a different compression methodology).

The header of a GDSII file optionally contains information about fonts, reference libraries, and other things. This information is saved as properties of the top-level cells derived from the file, i.e., those cells that are not used as subcells of another cell in the file. *Xic* does not use this information, but it will be put back into a GDSII file subsequently written by *Xic*, as other applications may need this information.

When reading GDSII or OASIS input, *Xic* will attempt to map the layer number and data type combinations found in the file to existing *Xic* layers, and if that fails a new *Xic* layer will be created. This is described in the section on GDSII layer mapping (14.6).

When reading CIF, layer names are matched to those defined in the current technology in a case-insensitive mode. This differs from native and CGX file types, which use case-sensitive matching. Layers found in the file which do not match any in the technology are created, using default parameters.

When a cell is written to disk, it is by default written in the format of origin, though a format change can be coerced in the **Save As** command by supplying a file extension. Explicit conversions can also be performed with the commands in the **Convert Menu**.

If a cell is opened for editing that contains empty cells, the user is given the option of deleting these references. If empty cells are found in the hierarchy, a pop-up appears, which allows their deletion. The cell names listed are those that for each mode (electrical and physical) the named cell either does not exist or has no content.

This test can be performed at any time with the **!empties** command. The test can be suppressed by setting the **Skip testing for empty cells** check box in the **Setup** page of the **Import Control** panel from the **Convert Menu**, or (equivalently) by setting the NoCheckEmpties variable.

8.2.3 Opening New Cells – Conflict Resolution

Xic keeps an internal database of all cells that have been used, by name. When a new file is opened for editing, it may contain definitions for cells with the same name as those already in memory. *Xic* provides several features for dealing with this situation when it arises.

The symbol table used to store cells can be changed. Creating and installing a new symbol table enables *Xic* to start with a fresh database, though the original database can be reinstalled at any time. There is no problem with cells of the same name existing in different symbol tables. The symbol tables are manipulated with the **Symbol Tables** panel from the **Cell Menu**. Symbol tables are useful for global context saving and switching, but since only one table can be installed at a time, it is generally not possible to access cells from different symbol tables simultaneously. Cells used in a hierarchy must exist in or be saved in the same symbol table.

When a file is being read from disk and a cell whose name conflicts with an existing cell in memory is encountered, a **Merge Control** pop-up will generally appear. This allows the user to choose whether or not to overwrite the physical and/or electrical part of the cell in memory. Press **Apply** to continue with the conversion. One must press **Apply** for each cell where there is a conflict, or press **Apply to Rest** to apply the present setting to the rest of the cells that clash. Dismissing the pop-up performs the same function as **Apply to Rest**. The pop-up is removed when all conversions are done.

If the NoAskOverwrite variable is set (with the **!set** command), or equivalently the **Don't prompt for overwrite instructions** button in the **Setup** page of the **Import Control** panel (from the **Convert Menu**) is active, no **Merge Control** pop-up will appear, and the default action will be used. The default action will also be used in non-graphics (server or batch) mode.

The default action can be specified by setting the NoOverwritePhys and/or the NoOverwriteElec variables, or equivalently by making a selection from the **Default when new cells conflict** menu in the **Setup** page of the **Import Control** panel. If no choice is made by any means, the default is to overwrite the cell in memory, both physical and electrical parts. The initial selections in the **Merge Control** pop-up will reflect the settings of the default action.

8.2.4 Object Tests

While a file is being read, tests for reentrant or otherwise strange polygons are normally performed. A polygon that is reentrant overlaps itself. This can be a problem since the polygon may be rendered differently on different CAD systems, as the presentation of the polygon may become ambiguous. The test is performed on physical data only. This adds a little overhead. The test is skipped if the boolean variable NoPolyCheck is set (with the **!set** command). This test can also be turned off from the **Setup** page of the **Import Control** panel.

There will also be a warning message added to the log if a polygon vertex list is modified by *Xic*. The checking function will remove duplicate, inline, and "needle" vertices. This does not change the shape of the polygon, but reduces complexity and memory use. If the file is written back to disk, the warnings will not reappear when reading the new file.

Similarly, wire objects are also tested for rendering difficulties. Wire objects consist of a vertex list, a width parameter, and an end style parameter. To render or otherwise process a wire, a polygon representing the actual shape has to be generated internally, making use of these parameters. With some parameter sets, this can be difficult or impossible. In addition, ambiguity arises between different tools in how (for example) acute angles are rendered, and how the "rounded" end style is implemented.

Wires that are impossible or difficult to render are logged. Wires that are impossible to render are never added to memory. Wires that are difficult to render are listed as "questionable" in the log file. These may or may not look "good" in the *Xic* display. It is possible that wires that look good in *Xic* will not be processed correctly in another tool, and vice-versa, so the user should be aware of the presence of these wires.

If when reading a file a warning message about "badly formed polygons" appears in the log file, here is how to proceed. Note the cell that contained the polygon, and edit it. Use the **!polycheck** command to select the bad polygons. The **Info** command in the **View Menu** can be used to obtain the vertex list. In many cases, the polygon will not cause problems, however it is wise to recreate one that is flagged as bad. The **Create Cell** command can be used to save the bad polygons to a separate cell for further inspection. A **!split** operation followed by a **!join** should effectively repair a degenerate polygon.

Similarly, there is a **!wirecheck** command that can be used to identify "questionable" wires in the current cell. To avoid problems down-stream, these should probably be converted to polygons. This can be done with **!split/!join**, or with the polygon creation command in the side menu.

By default, *Xic* checks for identical, coincident objects when reading input files, and prints a warning message in the log file if such objects are found. The **Duplicate item handling** menu in the **Setup** page of the **Import Control** panel can be used to set the action to perform on duplicates. Choices are no checking at all, warn only, or warn and remove duplicates.

8.2.5 The File Selection Panel

The **File Selection** panel allows the user to navigate the host's file systems, and select a file for input to the program.

The panel provides two windows; the left window displays the subdirectories in a tree format, and the right window displays a listing of files in a columnar form. The panel is similar in operation to the Windows Explorer tool provided by Microsoft.

When the panel first appears, the directories listing contains a single entry, which is shown selected, and the files window contains a list of files found in that directory. The tree "root" is selected by the application, and may or may not be the current directory. If the directory contains subdirectories, a small box containing a '+' symbol will appear next to the directory entry. Clicking on the '+' will cause the subdirectories to be displayed in the directory listing, and the '+' will change to a '−'. Clicking again on the '−' will hide the subdirectory entries. Clicking on a subdirectory name will select that subdirectory, and list its files in the files listing window. The '+' box will appear with subdirectories only after the subdirectory is selected.

Clicking on the blue triangle in the menu bar will push the current tree root to its parent directory. If the tree root is pushed to the top level directory, the blue triangle is grayed. The label at the bottom of the panel displays the current root of the tree. There is also a **New Root** item in the **File** menu, which allows the user to enter a new root directory for the tree listing. In Windows, this must be used to list files on a drive other than the current drive.

The **Up** menu is similar, but it produces a drop-down list of parent directories. Selecting one of the parents will set the root to that parent, the same as pressing the blue triangle button multiple times to climb the directory tree.

The **New CWD** button in the **File** menu allows the user to enter a new current working directory for the program. This will also reset the root to the new current working directory. The small dialog window which receives the input, and also a similar dialog window associated with the **New Root** button, are sensitive as drop receivers for files. In particular, one can drag a directory from the tree listing and drop it on the dialog, and the text of the dialog will be set to the full path to the directory.

The files listed in the files listing always correspond to the currently selected directory. File names can be selected in the files listing window, and once selected, the files can be transferred to the calling application. The **Go** button, which has a green octagon icon, accomplishes this, as does the **Open** entry in the **File** menu. These buttons are only active when a file is selected. One can also double-click the file name which will send the file to the application, whether or not the name was selected.

Files can be dragged and dropped into the application, as an alternative to the **Go** button. Files and directories can also be dragged/dropped between multiple instances of the **File Selection** panel, or to other file manager programs, or to other directories within the same **File Selection** panel. The currently selected directory is the target for files dropped in the files listing window. When dragging in the directory listing, the underlying directory is highlighted. The highlighted directory will be the drop target.

By default, a confirmation pop-up will always appear after a drag/drop. This specifies the source and destination files or directories, and gives the user the choice of moving, copying or (if not in Windows) symbolically linking, or aborting the operation.

In *Xic*, the variable NoAskFileAction can be set to skip the confirmation. This was the behavior in releases prior to 3.0.0, and experienced users may prefer this. However, some users may find it too easy to inadvertently initiate an action.

If the NoAskFileAction variable is set, the following paragraphs apply.

The drag/drop operation is affected by which mouse button is used for dragging, and by pressing the **Shift** and **Ctrl** buttons during the drag. The normal operation (button 1 with no keys pressed) for drag/drop is copying. The other options are as follows:

Operations	
Button 1	Copy
Shift-Button 1	Move
Control-Button 1	Copy
Shift-Control-Button 1	Link
Button 2/3	Ask

Above, "Ask" means that a dialog will appear asking the user what operation to perform. Options are `move`, `copy`, or (symbolically) `link`. Both the source and destinations are shown in the pop-up, and can be modified.

If a directory is the source for a copy, the directory and all files and subdirectories are copied recursively, as with the "`-R`" option of the Unix "`cp`" command.

Only one file or directory can be selected. When the operation is `copy`, the cursor icon contains a '+' in all cases. This will appear when the user presses the **Ctrl** key, if the underlying window supports a `move` operation.

The **File** menu contains a number of commands which provide additional manipulations. The **New Folder** button will create a subdirectory in the currently selected directory (after prompting for a name). The **Delete** button will delete the currently selected file. If no file is selected, and the currently selected directory has no files or subdirectories, it will be deleted. The **Rename** command allows the name of the currently selected file to be changed. If no file is selected, the name change applies to the currently selected directory.

The **Listing** menu contains entries which affect the file name list. By default, all files are listed, however the user can restrict the listing to certain files with the filtering option. The **Show Filter** button displays an option menu at the bottom of the files listing. The first two choices are "all files" and the set of extensions known to correspond to supported layout file formats. The remaining choices are editable and can be set by the user. The format is the same as one uses on a Unix command line for, e.g., the `ls` command, except that the characters up to the first colon (':') are ignored. It is intended that the first token be a name for the pattern set, followed by a colon. The remaining tokens are space-separated patterns, any one of which if matching a file will cause the file to be listed.

In matching filenames, the character '`.`' at the beginning of a filename must be matched explicitly. The character '`*`' matches any string of characters, including the null string. The character '`?`' matches any single character. The sequence '[...]' matches any one of the characters enclosed. Within '[...]', a pair of characters separated by '`-`' matches any character lexically between the two. Some patterns can be negated: The sequence '[^...]' matches any single character not specified by the characters and/or ranges of characters in the braces. An entire pattern can also be negated with '`^`'. The notation '`a{b,c,d}e`' is a shorthand for '`abe ace ade`'.

The **Relist** button will update the files list. The file listing is automatically updated when a new filter is selected, or when **Enter** is pressed when editing a filter string.

The files are normally listed alphabetically, however if **List by Date** is selected, files will be listed in reverse chronological order of their creation or last modification time. Thus, the most-recently modified file will be listed first.

The **Show Label** toggle button controls whether or not the label area is shown. The label area contains the root directory and current directory, or a file info string. By default, the label area is shown when the pop-up is created as a stand-alone file selector, but is not shown when the pop-up appears as an adjunct when soliciting a file name.

When the pointer is over a file name in the file listing, info about the file is printed in the label area (if the label area is visible). This is a string very similar to the "`ls -l`" file listing in Unix/Linux. It provides:

1. The permission bit settings and file type codes as in "`ls -l`" (Unix/Linux only).
2. The owner and group (Unix/Linux only).
3. The file size in bytes.
4. The last modification date and time.

While the panel is active, a monitor is applied to the listed files and directories which will automatically update the display if the directories change. The listings should respond to external file or directory additions or deletions within half a second.

The **File Selection** pop-up appears when the **File Select** button in the *Xic* **File Menu** is pressed. Variations of **File Selection** panel appear when the user is being prompted (from the prompt line) for a path to a file to open or write, such as for the commands in the **Convert Menu**. The **Open File** dialog is used when a path to a file to open is being requested. It is almost the same as the **File Selection** panel, except that selecting a file will load that path into the prompt line. The **Save File** dialog is used when the user is being prompted for the name of a file to save. This does not contain the list of files found in the other variations, but allows the user to select a directory.

8.3 The Save Button: Save Modified Cells

The **Save** button in the **File Menu** allows saving unsaved work to disk files, under the present file/cell name.

If there are cells in memory that have been modified, the **Modified Cells** pop-up will appear. This is the same pop-up that appears when exiting *Xic* if there are unsaved cells. It can also be invoked with the **!sv** command.

The pop-up displays a listing of all modified cells and hierarchies, each with a yes/no entry that can be toggled by the user to set whether the cell or hierarchy will be saved. The display has four columns. Column 1 gives the name of the cell, which for a hierarchy is the top level cell.

The second column is "`yes`" or "`no`". Clicking on this word will toggle between the two states. The buttons at the top of the panel will set the states of all of these words: **Save All** sets them to "`yes`", **Skip All** sets them to "`no`".

Initially, all normal cells in the listing will be set to "`yes`", meaning that all of the listed items will be updated on disk. If PCell submasters are being listed, then their initial state is "`no`", meaning that the master cell of a specific PCell instance and parameter set will not be written to disk. By default, the PCell sub-masters that are created in memory when a PCell is instantiated are not listed in the **Modified Cells** pop-up.

PCell sub-master cells are normally recreated in memory from the original parameterized cell definition when needed. However, there may be times when keeping a cache of PCell sub-masters is useful for performance reasons, or to export where the original PCell is not available or the format not supported.

If the boolean variable PcListSubMasters is set, then sub-masters created in memory for PCell instantiation will be listed in the **Modified Cells** pop-up.

The third column gives the type of file that will be created or updated. This entry is shown in color, and the color used for archives is different than the color used for native and other single-cell files.

`X`	*Xic* native
`B`	CGX
`C`	CIF
`G`	GDSII
`O`	OASIS
`A`	OpenAccess
`P`	PCell sub-masters (native)

If a cell was read from an OpenAccess library and modified, it will (by default) be saved to the same library. *Xic* can write only to OpenAccess libraries that were created by *Xic* or otherwise "branded" by *Xic* (with the **!oabrand** command). This should prevent unintentional overwriting of Virtuoso cells. Overwriting a Virtuoso cell from *Xic* will hopelessly clobber the cell for Virtuoso. Some day this may work, but for now expect the worst.

If saved, PCell sub-masters will be saved as native cell files in the current directory.

The fourth column is the full path name of the file that will be written if the second column is "`yes`". In the case of OpenAccess, this will be the library name.

Xic native cells are saved under their own name, in the directory containing the file the cell was read from, or the current directory if the cell was created within *Xic*. If a cell from an archive file was modified, the hierarchy is saved in the name of the original archive file, or the top-level cell name with an extension if the original file name is unknown. The file type is the same as the origin of the hierarchy. The **Save As** button can be used to save under a different name or file type.

In all cases, the previous version of an overwritten file is given a "`.bak`" extension and retained (any existing "`.bak`" file will be overwritten, however).

While the pop-up is visible, most other controls are inoperable. Pressing **Apply - Continue**, or deleting the window, will save the files marked "`yes`", retire the pop-up, and allow *Xic* to continue. Pressing the **ABORT** button will retire the pop-up and abort the present command.

8.4 The Save As Button: Save Cell, Renaming

The **Save As** button in the **File Menu** will save to disk the cell or hierarchy currently being edited, possibly under a new name or file type.

If editing a cell from the device library, the **Save As** command will bring up the **Library Cell Parameters** panel (see 8.5), which allows device defaults to be edited, and has provision for saving the cell into a device library file or as a native cell file.

Otherwise, the **Save File** dialog appears which provides an expandable and selectable tree representation of the directory structure, rooted in the directory where the file was originally read from, or the current directory. The name or path to the file can be modified on the prompt line, or directories can be selected from the pop-up which will modify the prompt line.

If the default is accepted, the cell or hierarchy will be saved in the format of origin: one of the archive formats, or native.

The response string actually supports syntax which provides coercion to another format, and other features. The general form of the response string is:

[*filetype*] *file_path* [*cellname*]

If the first word in the string is a recognized file format keyword, which is a known file format suffix **without** the period, output will be generated in that format. The following *filetype* keywords are recognized:

CGX	"`cgx`"
CIF	"`cif`"
GDSII	"`gds`", "`str`", "`strm`", "`stream`"
OASIS	"`oas`"
OpenAccess	"`oa`"
Native	"`xic`"

If the first word is not one of the recognized format keywords, then it is taken as a path to the output to produce. If this path has a file extension from the list above, meaning that the file name ends with a period followed by one of the words from the table, this will specify that format type for output. This does not apply to OpenAccess, however.

OpenAccess is available only if the plug-in was successfully loaded (see 2.11).

If the specified output format is one of the archive formats (CGX, CIF, GDSII, OASIS), then the entire cell hierarchy under the current cell will be saved in the output file produced.

If saving a hierarchy in CGX or GDSII format, the file name can be given an additional, final suffix "`.gz`", which will cause the file to be written in compressed (gzipped) format. These compressed files can be read into *Xic* directly, and can be uncompressed using the widely available GNU `gzip` or `gunzip` programs. Compression is supported for CGX and GDSII files only. The "`.gz`" suffix can be removed, if already present, to suppress compression.

If the file extension given is "`.xic`", then the current cell (not hierarchy) is saved in the file specified as a native cell file. The file, and the new cell name, will include the "`.xic`" extension. It is usually preferable to use the "`xic` *filetype* keyword to coerce native output to avoid changing the cell name.

There are a number of ways to save to native symbol files, as explained below. The general form is

[`xic`] [*word1* [*word2*]]

word1	**word2**	**description**
blank	blank	Save the current cell (only) as a native cell file in the current directory.
*	blank	A literal asterisk indicates to save all cells in the current hierarchy as native cell files in the current directory.
word	blank	If *word* is a path to an existing directory, save the current cell as a native cell file in that directory. If *word* is a path to an existing file, first move the existing file out of the way by giving it a `.bak` extension, then save the cell under the given file name. Otherwise, *word* is taken as a new name for the cell, which may contain a directory path. The native cell will be saved under that name.
word1	*word2*	The first word is taken as a directory path. This directory will be created if it doesn't exist, if possible. The second word is a new name for the cell. This must be a simple name, not a path. The current cell will be saved in the directory as a native cell file using the new name.
word	.	The first word is taken as a directory path. This directory will be created if it doesn't exist, if possible. The literal period as the second word indicates to save the current cell it the directory as a native cell file, using the present cell name. This form is useful to force creation of the directory.
word	*	The first word is taken as a directory path. This directory will be created if it doesn't exist, if possible. The literal asterisk as the second word indicates to save all cells in the hierarchy of the current cell, as native cell files in the directory.

The `xic` filetype specifier can be omitted if the source of the current cell is a native cell file. If omitted, in any case if the *word1* is a path to an existing directory (including "`.`" as the current directory), the "`xic`" is understood, and the behavior is as described in the table above.

To save to an OpenAccess library, the "`oa`" *filetype* **must** be given, any added file extensions are not recognized. The remainder of the line is interpreted as follows:

word1	**word2**	**description**
blank	blank	The current cell is written to the library named in the OaDefLibrary variable.
*	blank	If only an asterisk appears, the current cell and its hierarchy are written to the library named in the OaDefLibrary variable.
library	blank	If a single word is given, it is taken as the name of a library in which to save the current cell. If no such library exists, the user will be prompted to create it.
library	*cell*	If two words are given, the first word is taken as the library name as above. The second word is the name that the current cell will be saved under, thus the OpenAccess cell name can be different.
library	*	If an asterisk follows the library name, the current cell and its hierarchy will be written to the library.

When a file is read into *Xic*, the full path to that file is saved within *Xic*, and that file is the default written to during a save. The previous version of a file that has been overwritten is saved in a file in the same directory with the same name, but with a "`.bak`" extension added. Cells that are created within *Xic*, i.e., that do not have a known origin file, are saved by default in the current directory. This includes native-format versions of cells that were read in as part of an archive file.

8.5 The Save As Device Button: Editing Devices

The **Save As Device** button appears in the **File Menu** in electrical mode only. If the current cell is suitable as a device definition, meaning that the physical part is empty and there are no subcells, then the **Device Parameters** panel will appear. From this panel, the default device properties can be set, and the current cell saved as a device in either a file or an updated device library.

Devices in the device library can be edited, while in electrical mode, by simply giving the device name to the **Open** command or equivalent, and enabling editing mode with the **Enable Editing** button in the **Edit Menu**. When saving, with either **Save** or **Save As**, the **Device Parameters** pop-up will appear, as it will, of course, with the **Save As Device** button.

The panel will also appear in the **Save As** command if the name of the cell or file to save has been specified as the name of the device library file (default "`device.lib`"). Again, the cell must contain geometry appropriate for a device, i.e., no physical data and no subcells.

When creating a new device symbol, one can use an existing symbol from the device library as a starting point, and save under a new name. This will tend to keep the new device size and other characteristics similar to existing devices.

The remainder of this section describes the controls found in the **Device Parameters** panel.

The **subct** side-menu command is used to set the device connection points. The order of appearance on the SPICE line is the same as the numerical order in the marks shown in the **subct** command. The **subct** command creates the node properties required for electrical connection. At least one connection point is required, unless the **SPICE Prefix** begins with 'x' or 'X' (indicating a macro) in which case a connection point is not required. Thus it is possible for a macro, like a subcircuit, to connect to global nodes only.

The **Device Name** entry area contains the device (cell) name. This is arbitrary and can be changed, however a name must appear. This is the name by which the device is known to *Xic*, and the name that will appear in the device selection menu.

The **SPICE Prefix** is one or more characters that will be prepended to the device instance lines when a SPICE file is created. An entry in this field is usually mandatory. The pop-up will accept anything, however the first character should match the requirements of SPICE, which expects a certain key letter for each device, such as '`R`' for resistors (case independent). Additional characters can appear, and should be alphanumeric. An exception is the terminal device, which is not instantiated in SPICE, and must have a prefix starting with the character '`@`' for internal use by *Xic*. In *Xic*, the **SPICE prefix** for normal devices has no internal significance except as a unique identifier of that particular device, so the prefix should be unique in the device library file. The prefix is saved in a name property applied to the device.

If the prefix entry contains a second word "`macro`"', then the `macro` flag will be set in the name property. In this case, if the name prefix does not start with "`X`" or "`x`", *Xic* will prepend an "`X`" to instance calls, so that they are actually resolved as subcircuits. See the description of the name property for implications and use of this. A model property supplies the name of the SPICE `.subckt` that will be used. This must be supplied in the generated SPICE netlist by some means.

If the name prefix starts with "`X`" or "`x`", it is taken as a macro whether or not the keyword is given, to differentiate it from a normal subcircuit (which is not a "device"). A macro subcircuit is expected to reference a `.subckt` macro in the model library or another source. The name of the macro is given to instances of the device as a model property. A default model property can be supplied to the device. In the example in the provided device and model libraries, the name of the device is "`opamp`", and the model property is given as "`ua741`". There should be a file in the models subdirectories along the library search path, or an entry in the model library file, starting with "`.subckt ua741 ...`" and containing the subcircuit definition, terminated with "`.ends`". Note that subcircuits and models can be intermixed freely in the model files, but the reference names must be unique.

There is one special case: ground terminals. These have exactly one connection (a node property), and no other properties including a name property (prefix). If this matches the current cell, and a ground terminal in intended, then the **SPICE Prefix** should be left blank.

The **Default Model** and **Default Value** fields are optional for devices. Either one, but not both can be given, providing a default model name or default value to the device. If both are given, the **Default Value** entry will be ignored. These entries translate into model and value properties applied to the device. Instances will inherit which ever of these properties is given, but they can be changed on a per-instance basis.

If the device is a macro, i.e., the `macro` keyword is given or the SPICE prefix starts with '`x`' or '`X`', then the **Default Model** field is mandatory and contains the name of the subcircuit that will be instantiated. This name should be found in a `.subckt` line in the model library or elsewhere.

The **Default Parameters** field provides a default parameter set for the device or macro. The string can be any text relevant to the device in the context of SPICE, and will appear as a param property when the device is instantiated. This property can subsequently be changed in the instances.

The **Hot Spot** button, and associated menu and entry area, allows a branch property to be applied to the device. The branch property allows an internal value or function to be associated with a location in the schematic symbol, which can be clicked on in the drawing to obtain the values, after a simulation. For most devices, this will yield the current through the device. The branch property is "internal", meaning that it can not be changed in instances by the user.

The **Hot Spot** button will be active when the device contains a branch property. Pressing the button will create the property.

The branch property contains the hot spot coordinates, which are marked on-screen with a white cross when the **Hot Spot** button is active. While the **Hot Spot** button is active, clicking in the drawing will move the hot spot, and the white cross, to the button-down location. The user should click to locate the hot spot where desired in the drawing. In most of the devices in the supplied device library file, the hot spot is located on the '+' symbol that appears near the top device terminal.

The menu contains an orientation for the hot spot data. This is needed when the returned value is a current, and indicates the actual direction of positive current flow, relative to the device symbol. Typically, the two device terminals are oriented vertically, with the '+' associated with the top terminal, which would imply that the orientation choice should be "**Down**". If a scalar value is returned, so that there is no orientation, the correct choice would be "**none**". This selection will set the style and orientation of the plot trace marker applied when the hot spot is clicked on in the **plot** and **iplot** (electrical side menu) commands.

The text entry provides an expression for the value to be returned. The description of the branch property in D.3 describes this. This is the *string* part of the property description line, and may be empty for inductors and voltage sources.

The **No Physical Implementation** box should be checked if the device will never have a direct correspondence to geometry in the physical layout. This is true for example for voltage and current sources. Devices with this property set will not be considered in LVS testing and will never appear in netlists extracted from physical data. The device terminals will never appear in physical layouts. This will apply a nophys property to the device.

Once all needed fields have been filled in, the device can be saved. The **Save in Library** button will perform the following steps:

1. The device library file will be copied to the current directory, if it doesn't already exist in the current directory. If it does exist in that directory, the file will be copied and given a "`.bak`" extension.

2. The present device is written into the device library file. If the name already appears in the file, the existing device will be replaced. If the name does not appear, the device will be appended to the file.

   It is critical that the first line of a device description in the device library be a comment naming the device, in the form

   (`Symbol:` *devname*);

   When updating the library, the process looks for lines of this form. *Xic* will always add this line, but it may not be present if the file has been hand edited.

3. The modified device library is read back into *Xic*, and *Xic* is updated to use the new library.

4. The pop-up is retired, and a message indicates completion.

If, instead, it is desirable to avoid touching the device library but the user wishes to save the device, the **Save as Cell File** button can be used to save the device as a native cell file.

After saving, the device selection menus are updated, in case the device was saved to a location that was referenced in the device library, such as by a `Directory` keyword.

Warning: Be aware that it is not good to have cell files lying around that conflict with cells provided by the device library, as they can potentially cause trouble. Such files should be moved somewhere safe, at least out of the search path.

8.6 The Print Button: Print Control Panel

The **Print** button from the **File Menu** brings up the **Print Control Panel** for controlling hard copy plot generation. The panel supports a variety of printers and file formats through internal drivers.

While the **Print Control Panel** is visible, *Xic* is in “print mode” where the colors and other attributes of the main drawing window are set to those in force for the current print driver. The print driver is selected with the **Format** menu in the **Print Control Panel**.

Each print driver can have its own set of attributes and colors, which can be set from the technology file. Thus colors, fill, etc., can be set to provide best results from the driver. Changing the colors or attributes while in print mode will affect the setting for the current print driver only, and the original setting will be restored when print mode is exited. The settings applied to a driver are remembered the next time the driver is selected in print mode.

If, after setting up print driver-specific attributes and colors, the **Save Tech** button is used to generate a technology file, the file will contain the driver-specific information.

The driver-specific attributes include all of the settings from the **Main Window** sub-menu of the **Attributes Menu**, including all grid settings other than the spacing and snapping values. Grid spacing and snapping values carry over when switching to and from print mode. Individual layer colors, as well as the other attribute colors used in drawing windows, can be set for the driver with the **Color Selection** panel from the **Set Color** button in the **Attributes Menu**. Fill patterns are set with the **Fill Pattern Editor**, from the **Set Fill** button. Layer visibility can be set for the driver by clicking with mouse button 2 in the layer table. All of these settings apply only to the current print driver when in print mode, instead of the general screen display as when not in print mode.

Not all attributes will be recognized and used by all print drivers. In particular, the “line draw” drivers will typically ignore the fill pattern and simply draw an outline, though the HPGL and Xfig drivers have a means to use predefined fill patterns defined in the specific interface protocol. This can be set up in the technology file by use of the `HPGLfilled` and `XfigFilled` keywords, respectively.

When using the Image driver, a **No Backg** check box will appear above the driver selection menu. When checked, image files produced will use a transparent background, meaning that existing background will show through when the image is used in a document. This applies to PNG and other formats that support this feature. (Qt releazes only)

The temporary file produced may be quite large in some cases. This file is created in the `/tmp` directory by default. If this directory has insufficient disk space the XIC_TMP_DIR environment variable should be set to a path to a suitable directory.

8.6.1 Print Control Panel

The **Print Control Panel** is a highly configurable multi-purpose printer interface used in many parts of *Xic* and *WRspice*. This section describes all of the available features, however many of these features may not be available, depending upon the context when the panel was invoked. For example, a modified version of this panel is used for printing text files. In that case, only the **Dismiss**, **To File**, and **Print** buttons are included. Most of the choices provided by the interface have defaults which can be set in the technology file. The driver default parameters and limits are modifiable in the technology file. The **Print Control Panel** is made visible, and hardcopy mode is made active, by the **Print** button in the **File Menu**.

Under Windows, the **Printer** field contains a drop-down menu listing the names of available printers. The initial selection is the system default printer. This default can be set with the DefaultPrintCmd variable.

Under Unix/Linux, the operating system command used to generate the plot is entered into the **Print Command** text area of the **Print Control Panel**. In this string, the characters "`%s`"i will be replaced with the name of the (temporary) file being printed. If there is no "`%s`", the file name will be added to the end of the string, separated by a space character. The string is sent to the operating system to generate the plot.

The temporary file used to hold plot data before it is sent to the printer is *not* deleted, so it is recommended that the print command include the option to delete the file when plotting is finished. In *Xic* the RmTmpFileMinutes variable can be set to an integer to enable automatic deletion of the temporary file.

If the **To File** button is active, then this same text field contains the name of a file to receive the plot data, and nothing is sent to the printer. The user must enter a name or path to the file, which will be created.

Xic normally supplies a legend on the hardcopy output, which can be suppressed by un-checking the **Legend** check box. The legend is an informational area added to the bottom of a plot. In contexts where there is no legend, this button will be absent. In *Xic*, a legend containing a list of the layers is available. In *WRspice*, there is no legend.

The size and location of the plot on the page can be specified with the **Width**, **Height**, **Left**, and **Top/Bottom** text areas. The dimensions are in inches, unless the **Metric** button is set, in which case the dimensions are in millimeters. The **Width**, **Height**, and offsets are always relative to the page in portrait orientation (even in landscape mode). The vertical offset is relative to either the top of the page, or the bottom of the page, depending on the details of the coordinate system used by the driver. The label is changed from **Top** to **Bottom** in the latter case. Thus, different sized pages are supported, without the driver having to know the exact page size.

The labels for the image height and width in the **Print Control Panel** are actually buttons. When pressed, the entry area for height/width is grayed, and the auto-height or auto-width feature is activated. Only one of these modes can be active. In auto-height, the printed height is determined by the given width, and the aspect ratio of the cell, frame box, or window to be printed. Similarly, in auto-width, the width is determined by the given height and the aspect ratio of the area to print. In auto-height mode, the height will be the minimum corresponding to the given width. This is particularly useful for printers with roll paper.

The full-page values for many standard paper sizes are selectable in the drop-down **Media** menu below the text areas. Selecting a paper size will load the appropriate values into the text areas to produce a full page image. Under Windows, the **Windows Native** driver requires that the actual paper type be selected. Otherwise, this merely specifies the default size of the image.

Portrait or landscape orientation is selectable by the drop-down menu. In portrait mode, the plot is in the same orientation as seen on-screen, and in landscape mode, the image is rotated 90 degrees. However, if the **Best Fit** check box is checked, the image can have either orientation, but the legend will appear as described. If using auto-height, the legend will always be in portrait orientation.

When the **Best Fit** button is active, the driver is allowed to rotate the image 90 degrees if this improves the fit to the aspect ratio of the plotting area. This supersedes the **Portrait/Landscape** setting for the image, but not for the legend, if displayed.

The landscape mode is available on all print drivers. The behavior differs somewhat between drivers. The PostScript and PCL drivers handle the full landscape presentation, i.e., rotating the legend as well as the image by 90 degrees. The other drivers will rotate the image, however, the legend will always be on the bottom. In this case, the image may have been rotated anyway if the **Best Fit** button is active, and rotating provides a larger image. The landscape mode forces the rotation.

Xic provides a **Frame** button which allows a portion of the graphical display to be selected for plotting. This sets the view produced in the print, which otherwise defaults to the full object shown on-screen (the full cell in *Xic*). To set the frame, one uses the mouse to define the diagonal endpoints of the region to be plotted. This region will appear on-screen as a dotted outline box. Deselect the **Frame** button to turn this feature off, and plot the full object. In *Xic*, if the display contains transient objects such as rulers, DRC error indications, or terminals, it may be necessary to use the **Frame** command if these objects are not included in the cell bounding box. If the objects extend outside of the cell boundary, they may be clipped in the plot, unless the frame is used.

The available output formats are listed in a drop-down menu. Printer resolutions are selectable in the adjacent resolution menu. Not all drivers support multiple resolutions. Higher resolutions generate larger files which take more time to process, and may cause fill patterns to become less differentiable.

When a PostScript line-draw driver is selected, a **Line Width** numeric entry area appears, which can be used to set the width of the lines used for drawing. The value given is in points, a point being 1/72 of an inch. Different printers may respond to the specified width in different ways, depending on physical characteristics. The default, when the line width is set to 0, is to use the narrowest line provided by the printer. At times, using fatter lines improves visibility for presentation graphics and similar.

Pressing the **Print** button actually generates the plot or creates the output file. This should be pressed once the appropriate parameters have been set. A pop-up message appears indicating success or failure of the operation.

Pressing the **Dismiss** button removes the panel and takes *Xic* out of hardcopy mode. The same effect is achieved by pressing the **Print** button in the **File Menu** a second time.

8.6.2 The Format Menu: Hardcopy File Formats

The printing system for *Xic* and *WRspice* provides a number of built-in drivers for producing output in various file formats. In Windows, an additional Windows Native driver uses the operating system to provide formatting, thus providing support for any graphical printer known to Windows. The data formats are selected from a drop-down menu available in the **Print Control Panel**. The name of the currently selected format is displayed on the panel. In *Xic* only drivers that have been enabled in the technology file are listed (all drivers are enabled by default). The format selections are described below.

Except for the Windows Native driver all formatting is done in the *Xic*/*WRspice* printer drivers, and the result is sent to the printer as "raw" data. This means that the selected printer *must* understand the format. In practice, this means that the printer selected must be a PostScript printer, and one of the PostScript formats used, or the printer can be an HP Laserjet, and the PCL format used, etc. The available formats are listed below.

PostScript bitmap
: The output is a two color PostScript bitmap of the plotted area.

PostScript bitmap, encoded
: This also produces a two color PostScript bitmap, but uses compression to reduce file size. Some elderly printers may not support the compression feature.

PostScript bitmap color
: This produces a PostScript RGB bitmap of the plotted area. These files can grow quite large, as three bytes per pixel must be stored.

PostScript bitmap color, encoded
: This generates a compressed PostScript RGB bitmap of the plotted area. Due to the file size, this format should be used in preference to the non-compressing format, unless the local printer does not support PostScript run length decoding.

Postscript line draw, mono
: This driver produces a monochrome PostScript graphics list representing the plotted area.

PostScript line draw, color
: This produces an RGB color PostScript graphics list representing the plotted area.

HP laser PCL
: This driver roduces monochrome output suitable for HP and compatible printers. This typically processes more quickly than PostScript on these printers.

HPGL line draw, color
: This driver produces output in Hewlett-Packard Graphics Language, suitable for a variety of printers and plotters. In *Xic*, the fill patterns are defined in the technology file with the `HPGLfilled` keyword. Other fill pattern definitions are ignored. See the description of the HPGLfilled keyword in the technology file (section A.6) for more information.

Windows Native (Microsoft Windows versions only)
: This selection bypasses the drivers in *Xic* or *WRspice* and uses the driver supplied by Windows. Thus, any graphics printer supported by Windows should work with this driver.

  The Windows Native driver should be used when there is no other choice. If the printer has an oddball or proprietary interface, then the Windows Native driver is the one to use. However, for a PostScript printer, better results will probably be obtained with one of the built-in drivers. The same is true if the printer understands PCL, as do most laser printers. This may vary between printers, so one should experiment and use whatever works best.

In the Unix/Linux versions, selecting a page size from the **Media** menu will load that size into the entry areas that control printed image size. This is the only effect, and there is no communication of actual page size to the printer. This is true as well under Windows, except in the Windows Native driver. Microsoft's driver will clip the image to the page size before sending it to the printer, and will send a message to the printer giving the selected paper size. The printer may not print if the given paper size is not what is in the machine. Thus, when using this driver, it is necessary to select the actual paper size in use.

Xfig line draw, color

Xfig is a free (and very nice) drafting program available over the internet. Through the `transfig` program, which should be available from the same place, output can be further converted to a dozen or so different formats. In *Xic*, the fill patterns are defined in the technology file with the `XfigFilled` keyword. Other fill pattern definitions are ignored. See the description of the XfigFilled keyword in the technology file (section A.6) for more information.

Image: jpeg, tiff, png, etc

This driver converts into a multitude of bitmap file formats. This supports file generation only. The type of file is determined by the extension of the file name provided (the file name should have one!). The driver can convert to several formats internally, and can convert to many more by making use of "helper" programs that may be on your system.

Internal formats	
Extension	**Format**
`ppm, pnm, pgm`	portable bitmap (netpbm)
`ps`	PostScript
`jpg, jpeg`	JPEG
`png`	PNG
`tif, tiff`	TIFF

For the bitmap image formats, the driver resolution choice really doesn't change image resolution, but changes the size of the image bitmap in pixels. The image "resolution" is the number of pixels per inch in the image size entries. Thus, selecting a 4x4 inch image with resolution 100 would create a 400x400 pixel image. Note that selecting resolution 200 and size 4x4 would produce the same bitmap size as 100 and 8x8.

Under Microsoft Windows, an additional feature is available. If the word "`clipboard`" is entered in the **File Name** text box, the image will be composed in the Windows clipboard, from where it can be pasted into other Windows applications. There is no file generated in this case.

On Unix/Linux systems, if you have the open-source `ImageMagick` or `netpbm` packages installed then many more formats are available, including GIF and PDF. These programs are standard on most Linux distributions. The imsave system, which is used to implement this driver and otherwise generate image files, employs a special search path to find helper functions (`convert` from ImageMagick, the netpbm functions, `cjpeg` and `djpeg`). The search path (a colon-delimited list of directories) can be provided in the environment variable IMSAVE_PATH. If not set, the internal path is "`/usr/bin:/usr/local/bin:/usr/X11R6/bin`". The helper function capability is not available under Microsoft Windows.

If the **Legend** button is active, the image will contain the legend. If **Landscape** is selected, the image will be rotated 90 degrees.

The choice between PostScript line draw and bitmap formats is somewhat arbitrary. Although the data format is radically different, the plots should look substantially the same. A bitmap format typically takes about the same amount of time to process, independent of the data shown, whereas a line draw format takes longer with more objects to render. For very simple layouts and all schematics and *WRspice* plots, the line draw formats are the better choice, but for most layouts the bitmap format will be more efficient.

The necessary preamble for Encapsulated Postscript (EPSF-3.0) is included in all PostScript files, so that they may be included in other documents without modification.

8.7 The Files List Button: Path Files Listing Panel

The **Path Files Listing** panel lists the layout files found along the search path, including the files found through redirect files. The panel can be used to open files and cells for editing and placement, among other useful features. The file is brought up with the **Files List** button in the **File Menu**.

The panel contains a drop-down menu which has an entry for each directory in the search path, and each directory referenced in a redirect file. The main text area lists the files found in the currently selected directory.

File names are listed in columns. A character specifies the file type: "X" for *Xic*, "B" for CGX, "C" for CIF, "G" for GDSII, "O" for OASIS, and "L" for library files. Unrecognized file types are not listed. The directories are polled periodically, and the file listing is refreshed when changes are found. Unfortunately, this is not available under Windows 95/98/ME. In that case, resizing the window or popping the listing down then up again will refresh the listing.

The text area of the files listing is a drag and drop source and receiver. As a receiver, files or directories dropped in this area will appear in the directory that contains the listed files. By default, a confirmation pop-up will appear before any action occurs, but experienced users can disable this by setting the NoAskFileAction variable. See the description of the **File Selection** panel in 8.2.5 for the operations that can be performed via drag/drop. File names from the listing can be dragged into the drawing windows, which will load the file into the window.

A file can be selected by clicking on the name, and while selected it will be highlighted. When a file name is selected, the **Open**, **Place**, and **Contents** command buttons become active. These buttons are inactive (grayed) unless a file name is selected.

With a file name selected, pressing the **Open** button will load the file into the main window, as if the file was opened with the **Open** command in the **File Menu**. If the file is a library or has multiple top-level cells, a window appears which enables the user to make a selection to resolve the ambiguity. If the current cell is modified, the user will be given the opportunity to save it before switching to the new cell.

Similarly, pressing the **Place** button will load the top-level cell (after ambiguity resolution, if necessary) into the **Cell Placement Control** panel, from which it can be instantiated.

The **Contents** button brings up a panel which displays a listing of the cells found in the currently selected archive file, or a list of references if the selected file is a library. This button is enabled only when the selected file name corresponds to an archive or library (codes B, C, G, O, or L). The **Contents** button makes it possible to extract individual cells and subcells from an archive file, without having to load the whole file. It also provides access to the references contained within a library file.

The contents listing window contains **Open** and **Place** buttons. These buttons are normally grayed, but become active when a name is selected in the contents listing. Names are selecting by clicking with the mouse, as in the **Path Files Listing** panel.

Pressing the **Open** button will extract the named cell from the source file or library, along with its hierarchy, and load it into the main window. If the current cell is modified, the user will be given the opportunity to save it before switching to the new cell.

Similarly, pressing the **Place** button will load the selected cell into the **Cell Placement Control** panel, from which it can be instantiated.

The contents listing is a drag source for drag/drop. Names from the list can be dropped into a drawing window, with an effect similar to using the **Open** button. If a cell name from the contents list is dragged and dropped into a drawing window, that cell and its descendents will be extracted from the archive and displayed in the window.

When *Xic* is in CHD display mode, i.e., the **Display** button in the **Cell Hierarchy Digests** panel is active, the **Open** and **Place** buttons in the **Path Files Listing** and the contents window are not available. The **Place** buttons are not available in the *Xiv* feature set.

8.8 Cell Hierarchy and geometry Digests

Cell Hierarchy Digests (CHDs) are in-memory objects that map a cell hierarchy from a layout archive into a compact form, and are used to extract cell data. A "bare" CHD contains an offset into the original file for each cell, so that cell data are acquired by reading the original file.

The CHD facilitates extracting geometric information from the layout file on a per-cell basis, and is used internally during certain operations, including windowing, flattening, and empty cell filtering.

A CHD will contain physical and possibly electrical cell hierarchy data, as extracted from an archive file. Operations with a CHD that contains electrical data will either pass-through electrical data untouched, or strip it entirely. If the CHD is used to read into the database or to write a file, and there is no windowing or flattening, the electrical data will appear in the database or in the output file. If windowing or flattening is employed, only the physical data will be processed. The output will contain only the physical data.

A CHD facilitates random-access to cells within the file, which in general is a reasonably efficient process. However, if the source file is gzip-compressed (GDSII and CGX files only), random seeking can be a very slow process, as the decompression state must evolve from the beginning of the file. Seeking backwards requires rewinding the file and decompressing to the desired offset.

However, there is a random-access mapping option available, controlled by the setting of the ChdRandomGzip variable. This can speed random access into gzipped files, but requires some memory overhead. See the variable description for more information, this feature is not available in all *Xic* distributions.

The CHD is designed to minimize memory use, and allows processing of huge layout files that can not fit entirely in virtual memory in the normal database. Additional memory reduction is accomplished by saving cell instance lists in compressed form in memory. However, this may have a small computation overhead due to the required decompression before use. The ChdCmpThreshold variable can be used to turn off this compression, if speed is paramount and memory use is not an issue.

Optionally, a CHD can be linked to a companion data structure, called a Cell Geometry Digest (CGD). A CGD is a compact object that supplies cell geometry data. When a CGD is linked, cell geometry are obtained through the CGD (if present in the CGD), instead of from the original archive file. This can reduce access time considerably.

When using a CHD to access cell data, and the CHD has a linked CGD, and the cell data were previously removed from the CGD, the data will be obtained from the original layout file. Thus the

CGD can be used as a kind of cache.

There are three types of CGD:

1. The "memory" CGD saves all geometry data in memory. The geometry data are highly compressed, so that this makes sense even for very large layouts.
2. A "file" CGD instead stores offsets into a CGD file on disk. The disk file can also contain the CHD representation. This access method is not quite as fast as the in-memory variant, but is still generally much faster that reading the original layout file since 1) the data are highly compressed so fewer bytes are read, and 2) the data are sorted by layer so per-layer searches are more direct.
3. A "remote access" CGD obtains geometry data from a remote host which is running *Xic* in server mode. The CGD is a stub which links to a CGD in server memory, and data are returned via interprocess communication calls.

The three types indicate the creation mode of a CGD. In fact, the data access is specified on a per-record basis, so that a CGD could contain records of each type. The mixing of types, and specifically the ability to bring some records into memory (i.e., caching), will be more fully developed in future releases.

The CGD contains a reference count, which is incremented when the CGD is linked to a CHD, and decremented when unlinked. It is possible for a CGD to be used by multiple CHDs. It is not possible to destroy a CGD while the reference count is nonzero, i.e., when it is linked to a CHD.

In *Xic*, CHDs and CGDs are given access names, which are used to access the CHD or CGD in memory. These names are arbitrary but must be unique among the CHDs or CGDs. They may be assigned by the user or generated within *Xic*.

The **Cell Hierarchy Digests** panel, from the **Hierarchy Digests** button in the **File Menu** is the main entry point for creation and manipulation of CHDs. Similarly, the **Cell Geometry Digests** panel from the **Geometry Digests** button in the **File Menu** is the main entry point for CGD creation and manipulation. These two panels provide the GUI interface to CHD/CGD creation and manipulation.

In most if not all *Xic* commands that prompt for the name of a layout file, instead of a file name, the access name of an existing CHD can be given, or the name of a saved CHD file can be given. In the latter two cases, the command obtains geometric data through the CHD, which can be much faster, but operates as one would expect if directly giving the name of the referenced layout file.

However, a linked CGD provides only physical data, and properties and text labels are stripped.

8.9 The Hierarchy Digests Button: List Cell Hierarchy Digests

The **Hierarchy Digests** button in the **File Menu** brings up the **Cell Hierarchy Digests** listing of the Cell Hierarchy Digests (CHDs) currently in memory. A CHD is a compact representation of a cell hierarchy, which facilitates access to data on a per-cell basis. The CHD and companion Cell Geometry Digest (CGD) data structures provide a foundation for many of the operations in *Xic*, including windowing, flattening, and empty cell removal. An overview of CHD/CGD capabilities was provided in the previous section.

Each saved CHD has a unique but otherwise arbitrary access name. The access name is initially assigned by the user or generated by *Xic*.

The listing consists of the CHDs by access name. The middle column in the CHD listing will show the name of a linked CGD, if any. The right column lists the source file name and default top-level cell.

Most *Xic* commands that take a layout file path as input will accept a CHD access name. The command will operate with the data obtained through the CHD, which can be identical with that from the original layout file, but operations will in general proceed more quickly.

Clicking on one of the rows in the listing will select that CHD. The selected CHD is acted on by most of the command buttons arrayed along the top of the panel, which provide the following functions.

Add
: This button brings up the **Open Cell Hierarchy Digest** panel (described in 8.9.1) which allows a new CHD to be created and added to the list.

Save
: A CHD can be saved to a file, and recalled into memory later. This button produces the **Save Hierarchy Digest File** pop-up that solicits a file name/path into which a representation of the currently selected CHD will be saved. A previously saved CHD can be recalled with the **Add** button.

If the **Include geometry records in file** check box in the pop-up is checked, geometry records will be included in the file. These records are effectively a concatenation of a Cell Geometry Digest file representation. Layer filtering (see 14.5) can be employed to specify layers to include, through the layer filtering control group which is activated when including geometry.

The resulting file is a highly compact but easily random-accessible representation of the layout file. However, it does not include text labels, properties, or electrical data.

Delete
: The presently selected CHD is destroyed, after confirmation.

Config
: This brings up the **Configure Cell Hierarchy Digest** panel (described in 8.9.2) which enables configuration of the CHD. There are two attributes that may be configured: the assumed top-level cell in the hierarchy, and the linking of a CGD for geometry access. The pop-up provides control of these attributes.

Display
: When this toggle button is pressed, the main window and new sub-windows display the cell hierarchy in the CHD. Editing is not possible in any window in this mode, so the side menu becomes invisible. The display is very similar to that of the normal display mode. The usual zooming/panning, expansion, and other modes apply, though no selection operations are available. In CHD display mode, the **Edit**, **Modify**, **DRC**, and **Extract** menus are unavailable, and various other functions in the other menus are unavailable.

When the **Display** button is pressed, a small pop-up appears, which allows the user to select an area to display before the image is created, which is compute intensive and time consuming. The user should enter the center x and y and display width (in microns) of the region of the top-level cell to be displayed. Pressing **Apply** will create and display the image. Alternatively, the **Center Full View** button can be pressed to display the entire cell.

The features in the display are obtained through the CHD, and thus no additional memory is required than that used by the CHD itself. Since the CHD occupies a small fraction of the memory required to hold the originating layout file in the main database, very large files can be viewed, much larger than files viewed the normal way for a given amount of available system memory.

The row in the CHD listing that is currently being displayed is marked, by an "open" icon in Windows, or by a different background color. This display mode will persist as long as the **Display** button is active, whether or not the pop-up is visible.

The root cell in the display is initially the default cell from the CHD. This cell can be specified in the pop-up from the **Config** button. If no cell name is specified, the top-level cell in the CHD (a cell not used as a subcell within the CHD) with the lowest offset (there may be more than one) is assumed. If a Cell Geometry Digest (CGD) has been linked to the CHD in the configuration panel, the displayed geometry is obtained from the CGD. In this case, text labels, which are never included in the CGD database, are absent from the display.

Drag and drop can be used from the contents listing (below) to change the root cell in the display. This does not change the default cell of the CHD, and only applies to the display in the drop-target window.

Contents

This button brings up or updates a listing of the cells in the currently selected CHD. The cell names can be selected by clicking in the listing. Only cells which correspond to the current display mode (physical or electrical) are shown.

The contents listing pop-up contains **Info**, **Open**, and **Place** buttons, which are active when a name is selected. Pressing **Info** will display info about the selected cell, as saved in the CHD. Pressing **Open** will extract the selected cell and its hierarchy from the source file into the main database, and display it in the main window, as if opened with the **Open** button in the **File Menu**. Pressing **Place** will likewise extract the cell hierarchy, but load it into the **Cell Placement Control** panel for instantiation.

The contents listing is enabled as a drag source. If an item is dragged to a drawing window and dropped the following will happen. If the drop window is displaying the CHD (the **Display** button is active), the window display will become rooted in the dropped cell. Nothing new is read into memory. If the drop window is in normal display mode, the cell and its hierarchy will be read from the CHD's source into the main database, and the cell will be displayed. Note that this can cause out-of-memory problems if one isn't careful.

Cell

It is possible to create "reference cells" in the main database that reference the CHD. These cells are otherwise empty, but when placed in a layout, and the layout is saved to disk, the hierarchy from the CHD will be written into the output file. See 8.9.3 more more information about reference cells.

This can be used to assemble a top-level cell or reticle containing very large amounts of data, far more than can be kept in memory in the usual way.

Pressing the **Cell** button will solicit the name of the reference cell. This is the name of a cell found in the CHD, and will also be the name of the reference cell created in memory. The pop-up is initially loaded with the name of the default cell of the CHD, but another cell name can be dragged from the contents listing or entered manually.

Pressing **Apply** in the solicitation pop-up will create the reference cell in memory.

In normal editing mode, the reference cells can be placed in the normal way (though they appear to have no content – they display as an empty box). The reference cells can be saved as native cells, in which case they remain as reference cells, and can be loaded into *Xic* just as any native cell.

When a reference cell is written to an archive file such as GDSII or OASIS, the reference cell is replaced by the cell and its hierarchy, as extracted from the original layout file.

Reference cells cannot be flattened with the **Flatten** command, they will simply disappear.

Info

Pressing this button will bring up or update a window containing information about the currently-selected CHD.

? (quick info)
This button brings up "quick info" about the currently selected CHD, including the full path to the source file. The same information can be obtained from the **Info** button, but this is much more extensive and may take some time to compute. The quick info is instantaneous.

Help
This brings up the help window describing the **Cell Hierarchy Digests** pop-up.

The buttons and controls below the listing window provide general CHD-related functions, that do not make use of selections in the listing.

Use auto-rename when writing CHD reference cells
This mode applies when writing a cell hierarchy containing reference cells. A reference cell is a cell in memory that has no content of its own, but rather serves as a pointer to a cell hierarchy obtained through a CHD (Created with the **Cell** button described above). When such cells are encountered when writing a hierarchy from the main database, the reference cell is replaced with the hierarchy obtained through the referenced CGD.

When writing CHD reference hierarchies, there are two algorithms that can be employed that prevent writing duplicate cell names. When this check box is not checked, cells encountered with the same name as a cell previously written will be skipped, i.e., no new cell definition will be added to the output file, and all subsequent instances of the cell will call the existing definition.

When this box is checked, and a duplicate cell name is encountered, and the existing definition came from a different CHD, the name is changed and a new cell definition is added to the output file. References to the cell will call the cell by its new name. However, name clashes from equivalent CHD's will cause the new cell definition to be skipped, as in the default mode. An "equivalent CHD" can mean the same CHD in memory, or a different CHD but opened on the same file with the same aliasing.

This button tracks the state of the RefCellAutoRename variable.

Load top cell only
When a cell is brought into the main database through a CHD, if this box is checked:

1. Only that cell, and not its subcells, will be loaded into the main database. Any subcells of the cell become reference cells (see 8.9.3) in the main database.
2. The name of the cell will be added to the override table.

This allows editing of the requested cell, and when written to disk the complete hierarchy will appear, however loading the whole hierarchy into memory is avoided.

This check box tracks the state of the ChdLoadTopOnly variable.

Fail on unresolved
This check box tracks the state of the ChdFailOnUnresolved variable. When set, when using a CHD to access cell data and a cell is found that can't be resolved in the source file or through the library mechanism, the operation will halt with a fatal error. If not set, processing will continue, with the non-references either being ignored (e.g., when flattening), or converted to empty cells (when reading into the database), or propagated to output (when writing output), depending on the operation.

Use cell table
When checked, when a CHD is used to access cell data, cells found in the override table (see 8.9.4)

will override those in the source. Depending on settings, such cells may be effectively replaced by cells in memory, or simply skipped.

This check box tracks the state of the UseCellTab variable.

Edit Cell Table

This button displays the **Cell Table Listing** panel. This enables editing of the list of cell names that are treated specially during CHD file-access operations, the "override table".

Default Geometry Handling

This menu sets the default way to handle geometry records found when reading a saved CHD file. This mode will apply when a CHD file name is given as input for a command (which is generally possible for commands that are soliciting a layout file), and there is no specific means of controlling the geometry record processing.

There are three choices. The initial default is to create a memory-type CGD from the geometry records, and link it to the CHD. In this case, all geometry data will reside in memory, which makes sense even for very large designs as the data are highly compressed. The second option is to create a file-type CGD and link it to the CHD. In this type of CGD, geometry is obtained from the geometry records in the CHD file when needed, and does not reside in memory. The third option is to ignore the geometry records, and therefor not create a linked CGD. Geometry will be obtained from the original layout file in this case (the original layout file must still exist in the same location as when the CHD file was created).

8.9.1 The Open Cell Hierarchy Panel

This panel specifies a path to a layout or saved Cell Hierarchy Digest (CHD) file, from which a new CHD will be created in memory and added to the **Cell Hierarchy Digests** listing. The panel is brought up with the **Add** button in the **Cell Hierarchy Digests** panel.

The panel provides two separate "notebook" tabs that specify the type of file to read: layout file or saved CHD file. The notebook pages expose the controls applicable to the type of input, however either type of file can be entered in the entry area of either page. The tabs serve to simplify the panel.

All cell hierarchy data, both physical and electrical, will be extracted from a layout file. However, if the **LockMode** variable is set while in physical mode, the electrical data, if any, will be omitted. If the source is a saved CHD file, the CHD in memory will be recreated verbatim, ignoring current mode settings.

When the source is a layout file, systematic cell name modifications can be applied, if desired. This is sometimes useful for avoiding name clashes. If cell name modification is used, the modified names must be used when specifying a cell to the new CHD, the original cell names are not retained.

When reading a layout file, it is possible to save some statistical information in the CHD, regarding counts of the geometrical objects in the file. This information will increase the size of the CHD in memory, with the bottom selection requiring the most memory, the top selection the least. The information saved is counts of the number of boxes, polygons, and wires seen. The choices are:

no geometry info saved

Don't save any statistical information.

totals only

This is the default, the totals for the file will be available.

per layer counts
The total counts for the file will be available for each layer used.

per cell counts
The counts will be available for each cell in the file.

per-cell and per-layer counts
The counts will be available for each layer used in each cell.

This information will be printed in the **Info** window of the **Cell Hierarchy Digests** pop-up. The file totals are shown in the CHD info, which is shown when there is no selection in the **Contents** window. The per-cell counts are shown in the **Info** window when a cell name is selected in the **Contents** listing.

If the CHD is going to be used in an operation with layer filtering, it is recommended that **per-cell and per-layer counts** be selected, as this allows efficient removal of cells made empty by the layer filtering (see 14.10).

If the file name specified is a saved CHD file (previously created from the **Save** button in the **Cell Hierarchy Digests** pop-up), then the other entries (cell name mapping and geometry counts) are ignored. The cell name mapping is retained from the original CHD that was saved. The geometry counts are presently discarded when a CHD is saved.

If the CHD file being read contains geometry records, the processing of these records can be specified by the radio buttons in the **CHD file** page. There are three choices. The first option is to create a memory-type CGD from the geometry records, and link it to the CHD. In this case, all geometry data will reside in memory, which makes sense even for very large designs as the data are highly compressed. The second option is to create a file-type CGD and link it to the CHD. In this type of CGD, geometry is obtained from the geometry records in the CHD file when needed, and does not reside in memory. The third option is to ignore the geometry records, and therefor not create a linked CGD. Geometry will be obtained from the original layout file in this case (the original layout file must still exist in the same location as when the CHD file was created).

These options are identical to default options which can be set from the **Cell Hierarchy Digests** panel, but the present panel overrides the default setting and applies only to the current operation.

8.9.2 The Configure Cell Hierarchy Digest Panel

The **Config** button in the **Cell Hierarchy Digests** panel brings up the **Configure Cell Hierarchy Digest** panel, with which it is possible to change the default top cell of a Cell Hierarchy Digest (CHD), and to link a Cell Geometry Digest (CGD) which can accelerate geometry record access.

The present default top-level cell name is shown in the editable area near the top of the pop-up. In an unconfigured CHD, the default top-level cell is the first cell encountered in the layout file that is not used as a subcell by any other cell in the file. Any cell defined in the file can be assigned as the top-level cell of the CHD. In any operation involving the CHD when a top-level cell is not otherwise specified, the configured cell will be taken as the default.

To configure a new top-level cell, use the **Contents** listing of the **Cell Hierarchy Digests** panel, if necessary, to identify an alternate cell name. Note that this is the name after any cell name modification is applied. A cell name can be dragged from the contents listing and dropped in the entry area, or the name can be entered manually.

Pressing the **Apply** button in this group will complete the cell name configuration. The label of the **Apply** button will change to "`Clear`", and the controls in this group will be grayed. The label at the top of the panel will indicate that a top-level cell has been configured. Pressing **Clear** will un-configure the top-level cell, reverting to the default.

The **Last** button will recall the last cell name used, if any.

A Cell Geometry Digest can be linked to the CHD. In this case, geometrical data retrieved through the CHD will be obtained from the CGD, and not the original layout file. This linking can be accomplished, or removed, with the lower group of controls.

To link an existing CGD, one enters its access name into the **CGD name** entry area. This is the name shown in the first column of the **Cell Geometry Digests** listing. Pressing the **Apply** button in this group will perform the link, gray the entries, and the button label will change to "`Clear`". The label text at the top of the panel will indicate that the CHD is now configured "with geometry". Pressing the **Clear** button will reverse the process.

If the name in the **CGD name** entry area matches an existing CGD, that CGD will be linked, whatever the status of the **Open new CGD** check box. If **Open new CGD** is checked, and the **CGD name** is empty or a non-matching name, a new CGD will be created, and either saved under the name given, or assigned a new name by *Xic* if no name is given.

Pressing **Apply** when a new CGD is to be created will bring up the **Open Cell Geometry Digest** panel. This allows setting up parameters in the new CGD as needed. Pressing **Apply** in this panel will complete the operation, as reflected by the state shown in the **Configure Cell Hierarchy Digest** panel. The new CGD will be listed in the **Cell Geometry Digests** panel, if it is visible.

When a CGD is created in this manner, specifically for linking to a CHD, the new CGD will be automatically destroyed when unlinked from the CHD (or when the linking CHD is destroyed). One can see the CGD disappear from the **Cell Geometry Digests** panel when unlinked (**Clear** is pressed) in this case.

Please note that there is no way for the CHD to know whether the linked CHD applies to the same original layout file. Linking to a CHD produced from a completely different layout will "succeed", and there will be no errors even in use. As geometry is being read, if a cell is not found in the linked CGD, no geometry will be returned, and the cell will appear to contain no geometry. If is up to the user to make sure that CHD and linked CGD cell name spaces are compatible.

8.9.3 Reference Cells

Reference cells are "pseudo cells" which exist in memory or on disk as native cell files only. These cells contain no content, but instead reference another cell hierarchy. Reference cells have the same name as the top-level cell assumed in the referenced hierarchy. Reference cells can be used with physical layout data only.

When reference cells are placed in a layout, and the layout is written to an archive file format on disk, the reference cells are replaced with the hierarchy referenced.

Reference cells can be created from the **Cell Hierarchy Digests** panel, with the **Cell** button.

Here is an example to illustrate how reference cells may be created and used. Assume that we have a file named "`input.gds`" that contains a cell named "`input_top`.

From the **ell Hierarchy Digests** panel, the **Add** button is used to create a CHD for `input.gds`.

The resulting CHD is selected in the listing, and the **Cell** button is pressed. A pop-up will appear requesting the name for the cell. The default name is the default top-level cell for the CHD, or the configured name. If this is not our desired name "`input_top`", the text is changed accordingly, and

Apply is pressed. The reference cell will be created in memory (it will be listed in the **Cells Listing** panel).

If memory is tight, the CHD that was just created can be deleted. It will be recreated if necessary. The **Cell Hierarchy Digests** panel can be dismissed.

The user can view the new cell with the **Open** command. Note that it has a bounding box, but no content. Trying to modify the cell by adding a box, for example, will fail. Reference cells are immutable - meaning read-only.

The reference cell named "`input_top`" is ready to be placed into another hierarchy. One can begin editing a new cell, assume that it is called "`foo`". The user will be asked whether to save the previous (reference) cell. The reference cell can be saved as a native cell, however it is not possible to change the cell name. The cell can be saved in this manner if the user wants a copy which can be reused in the future. Incidently, it is possible to coerce saving of a reference cell to an archive format, as usual, in which case the new file will contain the referenced cell hierarchy.

The user should make sure that the current expansion level is set to 0. When editing "`foo`", the **place** button in the side menu can be used to place one or more instances of "`input_top`", perhaps using the **Current Transform** to rotate, mirror, or magnify the instances. This will be no different than placing normal instances. The bounding boxes of the newly placed cells will be visible, as normal, however if the expansion level is increased, the bounding boxes disappear and there is no visible indication of the newly place cells, except that the overall bounding box encompasses them. Again, the reference cells have no content.

The hierarchy under `foo` can be saved to an archive format in the usual manner, for example one can type "`sav`" in the drawing window or press the **Save As** button in the **File** menu. In response to the prompt, one can enter "`foo.gds`", for example, to produce a GDSII file, and press **Enter**. The user should then confirm saving to GDSII format at the confirmation prompt, and the file `foo.gds` will be created.

To have a look at the new GDSII file, the user can clear the database with the **Symbol Tables** pop-up or by typing "`!!Clear(0)`". Then, the **Open** command can be used to open `foo.gds`. The unexpanded display will look the same as before, but note now that when expanded, the contents of the cells are displayed, as obtained from the `input.gds` file, but this content is now included in `foo.gds`.

This procedure serves a similar purpose to the **Layout File Merge Tool** and the **!assemble** command, but is graphical and easier to perform. It enables assembling a higher-level layout file from lower-level component files. Since the component files don't have to be in memory, one can assemble huge layouts with a modest computer, using any of these techniques.

Reference Cell Structure

A reference cell is basically an empty physical native cell with a refcell property (property number 7150, as described in D.1). This property contains the information that ties the reference cell to a source and provides a bounding box. A complete example of a reference cell is shown below,

```
(Symbol asic2);
(xic 4.2.9 LinuxRHEL7_64 03/01/2016 04:36 GMT);
(PHYSICAL);
(RESOLUTION 1000);
( CREATED 3/1/2016 4:36:34, MODIFIED 3/1/2016 4:36:34 );
5 7150 filename="/usr/local/cad/layouts/asic2.gds" cellname=asic2
```

```
    bound=0.000,0.000,2328.100,2543.700;
9 asic2;
DS 0 1 1;
DF;
E
```

This reference cell is a stand-in for a cell named "asic2" which is found in the path given. Note the simple form of the cell, particularly realizing that the comment lines (enclosed in parentheses) are optional. It is completely feasible to create reference cells with a text editor. The only reason that the CHD was used is that it provides the correct cell bounding box. The bounding box is used in the display, but does not affect the actual location or size of the cell hierarchy when expanded.

8.9.4 The Cell Table Listing Panel: Set Override Cells

Whenever a Cell Hierarchy Digest (CHD) is used to access a cell hierarchy for any purpose *other* than to read the cells into the main database, a cell substitution mechanism can be employed. This mechanism is enabled by setting the UseCellTab variable, or the **Use cell table** check box in the **Cell Hierarchy Digests** panel.

Each symbol table contains a hash table for cell names, which is used as the "cell override table" when working with CHDs. The **Cell Table Listing** panel lists the cell names in this table, for the current symbol table. This panel is made available through the **Edit Cell Tab** button in the **Cell Hierarchy Digests** panel.

The names listed in the table are cells found in the global string table for cell names. This includes the names of cells read into memory, and the names of cells referenced in CHDs in memory. The names persist even if the corresponding cell or CHD is removed from memory, until a global clear is performed with the `ClearAll` script function.

The panel provides the following buttons to manipulate the table contents.

Add

The **Add** button produces an entry form that allows the user to enter a new cell name into the table. The name given must be that of a cell previously opened or referenced by a CHD, as explained above.

The listing window is also sensitive as a drop receiver, so that cell names can be dragged/dropped from other windows, such as the **Cells Listing** panel, or the **Contents** listing of the **Cell Hierarchy Digests** panel.

If a cell is read into the main database from a CHD, and the ChdLoadTopOnly variable is set, then the cell will automatically be added to the table.

The state of the ChdLoadTopOnly variable (set or not) tracks the state of the **Load top cell only** check box in the **Cell Hierarchy Digests** panel.

Remove

This button allows names to be removed from the table, individually.

Clear

The **Clear** button will remove all names from the table, after confirmation.

Override and **Skip**

These two mutually-exclusive selections set how entries in the table are to be used. When **Override** is selected, listed cells that exist in the main database will override the cell in the CHD, as described below. If an override cell does not exist in the main database in the current symbol table, the operation will fail with an error.

If **Skip** is selected, the cells will simply be skipped. This is applicable when writing an archive file via a CHD, in which case cell definitions for the override cells will not appear, however references to the cells will remain. The file will require the library mechanism or some other means of satisfying the references when the file is read. In this mode, it does not matter whether or not the named cells exist in the main database.

These two choices track the state of the SkipOverrideCells variable.

The table can also be maintained through use of the script functions described in F.4.3.

When a CHD is accessing cell data, if overriding is enabled and the cell name matches a name in the table, the CHD will access the cell in main memory and not from any other source. The contents of the cell will be streamed recursively, however only subcells with names that are also in the table will have cell definitions included. Subcells that are not included in the table should exist in the CHD, otherwise there will be an undefined cell in output.

Note that substituting cells will not prevent the CHD from outputting cells that, given the substitutions, are not used in the hierarchy. For example, suppose cell A in the CHD has an instance of cell B, and this is the only instantiation of B. Consider that A is overridden by a version that does not instantiate B. In the current release, the output file will contain B, as an unused cell (top-level).

As an example of how the override mechanism and related features can be used, imagine that we have a large GDSII layout file, and we would like to make a small modification to the top-level cell. Suppose that the file to too large to load into main memory in the usual way for editing.

The first step is to create a CHD for the file, using the **Cell Hierarchy Digests** panel from the **File Menu**. The **Add** button can be used to create the CHD, which will be listed on the panel.

Next, we grab the cell that we wish to modify into the main database. Select the CHD and press the **Contents** button in the **Cell Hierarchy Digests** panel. A listing of all cells in the file will appear, with the top-level cells listed first, with an asterisk.

Press the **Load Top Cell** button. With this button pressed, when a cell is opened in the main database from the CHD, only that cell, and not its complete hierarchy, will be opened in memory. This is important, since we know that the complete hierarchy of the cell we plan to edit will not fit in memory.

In the content listing, drag the name of the cell to be edited to the main drawing window and drop it there. The cell will be displayed, and is ready for editing. Note that, when unexpanded, all of the subcells appear normal, however when expanded, they disappear. The subcells are actually CHD reference cells, which have no content but serve as a pointer to the CHD when the subcell data is needed.

Once the appropriate changes have been made, there are two ways to save the modifications. The first way relies on the assumption we made earlier that the cell being edited is the top-level cell in the hierarchy. Since this is so, we could simply save the current cell as GDSII. When saving, the reference cells are expanded to the full hierarchy during writing.

The second method illustrates the use of the override cells. Press the **Edit Cell Tab** button to bring up the editor window for the override cell table. The cell of interest will already be listed, since it was automatically inserted when it was opened for editing from a CHD when the **Load Top Cell** button was active.

Press the **Use Cell Tab** button in the **Cell Hierarchy Digests** panel, which will enable use of the override table.

In the **Convert Menu**, press the **Conversion** button to bring up the **Conversion** panel. At the top of the **Conversion** panel, set the **Input Source** to **Cell Hier Name**, select the **GDSII** output format tab, then press the **Convert** button. When prompted, give the name of the CHD we created, from the **Cell Hierarchy Digests** panel, it will be something like "CellHier1". Then, give the name of a GDSII file to create. The new file will contain the modifications we performed.

8.10 The Geometry Digests Button: List Cell Geometry Digests

This panel, brought up with the **Geometry Digests** button in the **File Menu**, provides a list of Cell Geometry Digests currently in memory. A Cell Geometry Digest (CGD) is a per-layer/per-cell database of highly compacted representations of cell geometry. Logically, a cell name and layer name are passed to the database, which returns a data block which when expanded yields a representation of the geometry on the given layer in the given cell. The database contains no information about cell instances, and text labels and object properties are excluded.

This is basically a companion to the Cell Hierarchy Digest (CHD), which contains hierarchy information but no geometry information. The two data types together provide complete physical information about the file.

A CGD can be linked to a CHD. After linking, the CHD will retrieve needed geometrical information from the linked CGD, rather than from the original layout file. This can be faster, since the CGD geometry data may be in memory, and are sorted by layer and compacted. Even with all geometry data residing in memory, the combined size of the CHD/CGD structures is still much smaller than the memory required for loading the original layout file into the main database in the normal way. The main database, however, provides the spatial sorting for fast access of objects at a given location, which is absent in the CHD/CGD combination.

Each saved CGD is given a unique but otherwise arbitrary name, which is used to access the CGD. The CGDs presently in memory are listed by name, and can be selected by clicking.

The listing contains a middle column labeled **Type, Linked**, which will contain **Mem**, **File**, or **Rem** indicating the geometry storage type of the CGD. This will be followed by **yes** if the CGD is linked to a CHD. An asterisk '*****' will follow **yes** if the CGD will be destroyed when unlinked from its CHD. The right column contains the source file name, if any. The **Info** button will provide more information about the CGD, including the full path to the source file.

The selected CGD is used as input for operations initiated by the row of buttons arrayed across the top of the panel. These buttons are:

Add
: This button brings up the **Open Cell Geometry Digest** panel, from which a new CGD can be created and added to the list (see 8.11).

Save
: The currently selected CGD can be saved to a file, for later recall. This button brings up a pop-up which solicits a name for this file. Pressing **Apply** will save the selected CGD to a disk representation in the given file path. A previously saved CHD can be recalled into memory from the panel brought up by the **Add** button.

Delete
: This will destroy the selected CGD, after confirmation. Only CGDs that are not currently linked

to a CHD can be destroyed.

Contents
This will pop up or update a listing of the cells found in the selected CGD. With a name selected, the **Info** button becomes active. Clicking **Info** will pop up or update another window, which lists the layers used in the selected cell. Only layers that have associated geometry are saved in the CGD. Each layer is listed with two numbers, representing the size of the compressed data stream for the layer ('c') and the uncompressed size ('u'). These aren't particularly useful to the user, but do give some indication of how much geometry is associated with each layer. Beware, however, that gigabytes of replicated features may be represented by only a few bytes.

Info
This button pops up a window listing information about the selected CGD. The information includes the type of CGD, and other parameters such as memory use, cell count, etc.

8.11 The Open Cell Geometry Digest Panel

This panel is used to create a new **Cell Geometry Digest** in memory, which is added to the listing in the **Cell Geometry Digests** panel. This panel is brought up with the **Add** button in the **Cell Hierarchy Digests** panel.

There are three "notebook" tabs that correspond to the three types of CGD. Each corresponding page contains controls for setting the parameters appropriate for the selected CGD type.

in memory
The **in memory** tab corresponds to a "memory" CGD. This type of CGD saves all geometry data in memory. The geometry data are highly compressed, so that this makes sense even for very large layouts.

The source from which to create the CGD is entered into the entry area at the top of the page. The source can be one of the following:

1. A path to a layout (archive) file.
2. The access name of a CHD already in memory.
3. A path to a saved CHD file.
4. A path to a saved CGD file.

If the source is a layout file, one can apply layer filtering as the file is being read. It is also possible to apply cell name mapping. If mapping is employed, layer data are accessed via the modified cell names. If the CGD is to be linked with a CHD, the cell name mapping, if any is used, should be the same when creating the CHD and the CGD. The control groups below the entry expose the layer filtering and cell name mapping capabilities.

If the source is a CHD access name, or a CHD file, the cell name mapping is automatically set to the same as was used in creating the CHD. The layer filtering is available if the source is a CHD access name, or if the source is a CHD file saved without geometry records (with the **Save** button in the **Cell Hierarchy Digests** panel). If the source is a CHD file containing geometry records, the CGD uses those geometry records verbatim.

If the source is a saved CGD file (from the **Save** button in the **Cell Geometry Digests** panel), the CGD will import this file verbatim.

file reference

The **file reference** tab corresponds to a "file" CGD. This type of CGD stores offsets into a CGD file on disk. The disk file can potentially also contain a CHD representation. This access method is not quite as fast as the in-memory variant, but is still generally much faster that reading the original layout file since 1) the data are highly compressed so fewer bytes are read, and 2) the data are sorted by layer so per-layer searches are more direct.

This page consists of an entry area, into which a source is entered. The source can be either a path to a saved CGD file, or to a saved CHD file that contains geometry records. In either case, the new CGD is created to reference the geometry data by offset into the source file.

During its lifetime, this type of CGD maintains an open file descriptor to its source file. Although it is not likely, it may be possible to hit a system limit for open file descriptors if too many file CGDs are simultaneously open.

remote server reference

The **remote server reference** tab corresponds to a "remote access" CGD. This type of CGD obtains geometry data from a remote host which is running *Xic* in server mode (see 4.5). The remote access CGD is a stub which links to a CGD in server memory, and data are returned via interprocess communication calls.

This page provides separate entry areas for the host name, port, and remote CGD access name. These correspond to the remote host running the *Xic* server, which must have a CGD in memory (of any type). The new CGD will transparently link to the remote CGD, under a local access name.

The **Host name** entry must contain the network host name of the machine running the server. The **Port number** is optional, if not specified the port used defaults to 6115, which is the IANA registered port number for the "`xic/tcp`" service. If the server is for some reason using a different port number, that same port number must be entered. The access name of the CGD to reference on the server must be entered into the **Server CGD access name** entry area.

During its lifetime, this type of CGD maintains an open socket to the server. Since the number of connections is limited, it is best to free this type of CGD as soon as possible.

Below the notebook area is an entry for access name. This is the name under which the new CGD will be listed in the **Cell Geometry Digests** panel. A default is provided that is guaranteed not to conflict with an existing CGD.

The user can specify an access name. If the name is in use by an existing CGD, and the existing CGD is not linked to a CHD, it will be destroyed, and the new CGD will be created and saved under the same name. However, if the existing CGD is linked, it cannot be destroyed, and the CGD creation will fail with an error message.

When the **Apply** button is pressed, if all goes properly the source will be processed, the new CGD will be created, and added to the list in memory under the access name given.

8.12 The Libraries List Button: List Open Libraries

The **Libraries List** button in the **File Menu** brings up the **Libraries** panel, which displays a listing of libraries found along the present search path. To speed the search, only files with a "`.lib`" extension are checked for the library keyword at the top of the file, so library files that do not have a "`.lib`" extension will not appear in this list. The first column in the listing contains an icon which indicates whether the library is open or closed.

Open libraries are searched to resolve cells when a layout file is being read. Closed libraries are ignored. A library is opened if it is ever listed in a content window from the **Path Files Listing** panel, or if a cell from that library is ever directly opened, such as by giving "*/path/library cellname*" to the **Open** command in the **File Menu**, or if opened with the **Open/Close** button (see below).

Libraries are an important component of the *Xic* cell resolution capability. Immediately after an archive file has been read into the main database, the new hierarchy is traversed to identify cells that are referenced in the hierarchy but were not defined in the file. First, the open libraries are searched, and if an unresolved cell name matches a name in a library, the cell is read into memory through the library. The library file itself is usually only an indirection mechanism, with the actual cells saved in another archive file, or as native cell files, though it is also possible to define inline cells in the library file.

If a cell is not resolved in the open libraries, then the search path is traversed for a native cell file that matches the cell name. If one is found, it is read into memory. If not found, the unresolved cell becomes an empty cell, and will otherwise behave normally in the database. A warning will be issued in the log file when a cell is found to be unresolved.

The library mechanism is also available when a Cell Hierarchy Digest (CHD) is used for file access. If the archive file source for the CHD contained unresolved references, the CHD will likewise have unresolved references. These cells can be resolved when reading with the CHD if they match an open library reference to a cell in an archive file. Presently, native and inlined cells can not resolve CHD references, except when reading into the main database.

By default, a cell that can't be resolved through a library is not an error, it will be handled appropriately. Processing will continue, with the non-references either being ignored (e.g., when flattening), or converted to empty cells (when reading into the database), or propagated to output (when writing output), depending on the operation.

However, if the **Fail on Unresolved** button in the **Cell Hierarchy Digests** pop-up, or equivalently the **ChdFailOnUnresolved** variable is set, an unresolved cell will halt the operation with a fatal error.

When reading a library cell into memory, the hierarchy under the cell will also be read, unless the subcell name already exists in memory in which case that subcell will not be read.

Cells read through the library mechanism have two internal attribute flags set, which affect their behavior. First, the LIBRARY flag will, by default, prevent the cell from being written when a hierarchy containing the cell is written to an archive file. This means that the file will not be self-contained, and will require the presence of the (open) library to completely resolve all cells. Second, the IMMUTABLE flag is set, which prevents a cell from being modified or renamed. Thus, library cells by default can not be edited when opened in this manner.

The flags can be switched on and off for any cell with the **Set Cell Flags** panel from the **Flags** button in the **Cells Listing** panel.

Libraries are listed and searched in the order opened, and shown in the listing. When resolving a reference, the first match will apply, superseding any later entries. The libraries can be selected by clicking on the entries. When a library is selected, the **Open/Close** and **Contents** buttons become enabled, which will act on the selection. The selection has no other purpose.

The **Open/Close** button toggles the open state of the selected library. The **Open/Close** button is active when a library is selected in the **Libraries** panel. Without a selection, the button is grayed. Clicking the open/closed folder icon in the selected row will have the same effect as pressing the button. Closing a library merely removes it from the search list, and any cells in memory from the library remain.

The **Contents** button is also activated when a library is selected in the **Libraries** panel. Pressing **Contents** will pop up a listing of the contents of the selected library. The entries can be cells, archives, or other libraries. The contents items can be selected by clicking on the names. When a selection is active, the **Open** and **Place** buttons become active. The **Open** button will load the selected cell into the main window. The **Place** button will pop up the **Cell Placement Control** panel, loaded with the selected cell, with which the cell can be instantiated. If the selected item is another library or an archive file, an intermediate ambiguity resolution pop-up will appear, and the user must select a cell to edit or place.

The above is manifestly true only if the referenced cell is in an archive file. A native cell will always be superseded by an inlined cell of the same reference name found earlier in the library search. Also, the NoReadExclusive and AddToBack variables will affect cell name resolution as in a normal open.

The **No Overwrite Lib Cells** button tracks the state of the NoOverwriteLibCells variable. By default, cells in memory that were read from a library can be overwritten by cells of the same name subsequently read into memory from an archive or native cell file. If this button is set, library cells (with the LIBRARY flag set) in memory will not be overwritten.

The contents listing is a drag source for drag/drop. Names from the list can be dropped into a drawing window, with an effect similar to using the **Open** button.

When *Xic* is in CHD display mode, i.e., the **Display** button in the **Cell Hierarchy Digests** panel is active, the **Open** and **Place** buttons in the contents window are not available. The **Place** button is not available in the *Xiv* feature set.

8.13 The OpenAccess Libs Button: List OpenAccess Libraries

The **OpenAccess Libs** button will appear in the **File Menu** only if the OpenAccess plug-in has been loaded, in which case there is a connection to an OpenAccess database on the current computer. Pressing the **OpenAccess Libs** button brings up the **OpenAccess Libraries** panel. The release number of the OpenAccess database software in use is shown in the panel above the listing of available libraries. The listing displays the names of libraries specified in the OpenAccess `lib.defs` or `cds.lib` file.

Similar to the **Libraries List** panel, the first column in the listing contains an icon which indicates whether the library is open or closed. The comments in that description apply to OpenAccess (OA) cells opened in this manner as well. However, it is possible to list the content of OA libraries whether or not they are open. Regular libraries must be open for the contents to be listed.

Open libraries are searched to resolve cells when a layout file is being read into *Xic*. Closed libraries are ignored in this case. However, direct references to an OA library from an OA cell are always "open". The open and closed status is toggled by the **Open/Close** button in the panel, which acts on the entry which has been selected by clicking on it.

The **Open/Close** button toggles the open state of the selected OA library. The **Open/Close** button is active when a library is selected in the **OpenAccess Libraries** panel. Without a selection, the button is grayed. Clicking the open/closed folder icon in the selected row will have the same effect as pressing the button. Closing a library merely removes it from the search list, and any cells in memory from the library remain.

The second column in the listing indicates whether or not the library is writable from *Xic*. By default, libraries created in *Xic* are writable from *Xic*, other libraries are not. This prevents, for example, Virtuoso cells from being overwritten from *Xic*, which could cause loss of data (putting it mildly). The writability of the currently-selected library can be toggled with the **Writable Y/N** button. Clicking on the **Y** or **N** in the selected line will toggle the state, as if the button was pressed. Library writability can also be set with the **!oabrand** command.

The **Contents** button is also activated when a library is selected in the **OpenAccess Libraries** panel. Pressing **Contents** will pop up a listing of the cells in the selected OA library. The contents items can be selected by clicking on the names. When a selection is active, the **Open** and **Place** buttons become active. The **Open** button will load the selected cell into the main window. The **Place** button will pop up the **Cell Placement Control** panel, loaded with the selected cell, with which the cell can be instantiated.

One can specify whether to read and write physical or electrical data from OpenAccess, or both, with the **Data to use from OA** radio button group. These buttons track the OaUseOnly variable. If this variable is set to “1” or any word starting with “p” or “P”, only physical data will be converted. If set to “2” or any word starting with “e” or “E”, only electrical data (schematic and symbol) will be converted. If set to anything else or not set, both physical and electrical data will be converted.

The restriction applies to conversion to and from OpenAccess, by any method in *Xic*.

When a cell is read into *Xic* from OA, the OA “layout” view is read as the physical cell data, the “schematic” view is read as the schematic data, and the “symbol” view is read as the symbolic representation. These need not all exist. The same view names apply when writing data to OpenAccess.

These view names are the defaults, as used by Cadence Virtuoso. However, any of the OaDefLayoutView, OaDefSchematicView, and OaDefSymbolView variables can be defined to provide alternate default view names.

When reading electrical info into *Xic*, a simulator-specific view is used for obtaining CDF (component data, from Virtuoso) parameters and properties. By default, this view is named “`HspiceD`”, but another view can be chosen by setting the variable OaDefDevPropView. The default choice provides compatibility with Hspice, and therefor *WRspice* in fair measure. If is not an error if no `HspiceD` views are found.

These four variables have corresponding entry areas in the **OpenAccess Defaults** panel brought up with the **Defaults** button. The text of the variables and entry areas track one another.

The contents listing is a drag source for drag/drop. Names from the list can be dropped into a drawing window, with an effect similar to using the **Open** button.

When OpenAccess if available, the **Open** command and similar, when prompting for the name of a file or cell to load, will recognize an OpenAccess library name followed by a cell name (two space-separated words).

When *Xic* is in CHD display mode, i.e., the **Display** button in the **Cell Hierarchy Digests** panel is active, the **Open** and **Place** buttons in the contents window are not available. The **Place** button is not available in the *Xiv* feature set.

The **Create** button allows a new library to be created. When pressed, a pop-up appears, requesting a name for the library, which can be any name allowed by OpenAccess. Pressing the **Create** button in the pop-up will create the library, and its name will appear in the listing. The new library has write permission from *Xic*. It will attach the default technology if set, otherwise there is no technology associated with the new library, the user will probably need to use the **Tech** button to either create a local tech database in the library, or link to the tech database in another library. New libraries can also be created with the **!oanewlib** command.

The **Tech** button, which is un-grayed when a library is selected, brings up the **OpenAccess Tech** panel described in 8.14. This panel allows control of the technology database associated with the library. The **!oatech** command can also be used to set the technology database.

The **Defaults** button displays the **OpenAccess Defaults** panel described in 8.15, from which some parameters used by the OpenAccess interface can be defined.

The **Destroy** button is un-grayed when a library is selected. When pressed, and after confirmation, the selected library and all of its content will be destroyed. Presently, the library is removed from the `lib.defs` file, but not otherwise touched. To reclaim the disk space used by the library, the user can manually delete the corresponding directory. The **!oadelete** command can also be used to delete libraries, and to delete cells in libraries.

8.14 The OpenAccess Tech Panel

The **OpenAccess Tech** panel is brought up with the **Tech** button in the **OpenAccess Libraries** panel. This panel is only available when an OpenAccess database is being accessed with the plug-in, in which case the **OpenAccess Libs** button appears in the **File Menu**.

In OpenAccess, every library is generally required to have an associated technology database. The technology database contains information about layers, physical attributes, design constraints, etc., similar to the *Xic* technology file. The database can either be "attached" or "local". When attached, it references the technology database from another library. If local, the library contains its own private technology database, to which other libraries can attach. This panel controls the technology database for the library initially selected in the **OpenAccess Libraries** panel.

The settings indicate the current status of the library. The top line contains buttons and an entry area that control attached technology. In libraries containing user cells, it is most common that an attachment is used, typically to a library supplied by the foundry in a process design kit. In a typical situation, an organization may make use of a single foundry process for several users and projects. It is likely then that all of the user/project libraries attach to the one foundry library. In this case, the **Default Tech Library** in the **OpenAccess Defaults** panel or equivalently the ¡a OaDefTechLibrary variable can be set to the name of the foundry library. Then, new libraries will automatically attach this library, and the user will never have to use the **OpenAccess Tech** panel.

If a database is currently attached. the **Unattach** button will be un-grayed, and the name of the attached library will appear in the status area, just above the **Dismiss** button. Pressing **Unattach** will (you guessed it) unattach the referenced database, and the **Unattach** button will become grayed. One can reattach, or attach the technology from a different library, by entering the name of the target library and pressing the **Attach** button. The **Default** button will enter the default tech library name (if any) or the previous attachment name (if any) into the text entry area when pressed, and the entry area is not grayed.

If there is no attachment, then the **Create new Tech** button will be un-grayed, along with the **Attach** button. Pressing **Create New Tech** will create a new local technology database. The **Attach** button will be grayed, as it is not possible to have an attached database if a local database is present. The **Destroy Tech** button becomes un-grayed, and will destroy the local database when pressed.

The **!oatech** command can perform may of the same operations.

8.15 The OpenAccess Defaults Panel

The **OpenAccess Defaults** panel is brought up with the **Defaults** button in the **OpenAccess Libraries** panel. This panel is only available when an OpenAccess database is being accessed with the plug-in, in which case the **OpenAccess Libs** button appears in the **File Menu**.

Each of the entry areas tracks a variable that is used to set default behavior in the interface to the OpenAccess database.

Library Path
: This entry tracks the setting of the OaLibraryPath variable. It can be set to a full path to a directory. If a library to be accessed is not listed in the `lib.defs` (or `cds.lib`) file, the system will look for the library as a subdirectory of the directory path given. This allows use of OpenAccess libraries that are hidden from other tools.

Default Library
: This tracks the setting of the OaDefLibrary variable. It can be set to a library name found in the `lib.defs` (or `cds.lib`) file, or to a subdirectory of the **Library Path** if any. This will be used when reading from or writing to the OpenAccess database, if the library name is not otherwise specified.

Default Tech Library
: This tracks the setting of the **OaDefTechLibrary** variable. It can be set to a library name found in the `lib.defs` (or `cds.lib`) file, or to a subdirectory of the **Library Path** if any. When a library is created, it will attach the technology database associated with the library name entered, if any. If the named library has an attached technology, the same attachment will be applied to the new library. Otherwise, the new library will attach the local technology database of the named library.

Default Layout View
: This tracks the setting of the OaDefLayoutView variable. It specifies an alternate view name for physical layout data. If not specified, the default layout view name is "`layout`".

Default Schematic View
: This tracks the setting of the OaDefSchematicView variable. It specifies an alternate view name for electrical schematic data. If not specified, the default schematic view name is "`schematic`".

Default Symbol View
: This tracks the setting of the OaDefSymbolView variable. It specifies an alternate view name for electrical symbol data. If not specified, the default symbol view name is "`symbol`".

Default Properties View
: This tracks the setting of the OaDefDevPropView variable. It specifies an alternate view name for the simulator-specific view which (if present) provides values for certain device properties (from the Common Design Framework (CDF) database). If not specified, the default simulator view name is "`HspiceD`". This specifies Hspice compatibility in Virtuoso, which is a good match for *WRspice*.

Dump CDF files while reading
: This check box tracks the set/unset status of the OaDumpCdfFiles variable. When checked, when a parameterized cell is opened in OpenAccess, the Common Design Framework (CDF) data for the cell will be dumped to a file in the current directory. The file name is the cell name with a "`.cdf`" extension. This is for development/debugging.

8.16 The Quit Button: Exit *Xic*

Pressing the **Quit** button in the **File Menu** will exit *Xic*, after confirmation is there is unsaved work.

If there are modified cells, the pop-up described for the **Save** command appears. This displays a list of the cells and hierarchies that have been modified, and allows the user to save them.

Chapter 9

The Cell Menu: *Xic* Cell Navigation and Information

The **Cell Menu** contains the **Push**/**Pop** commands that enable pushing the viewing/editing context into the hierarchy, and returning. Other commands provide information about cells and allow other manipulations.

In *Xic*, there is a notion of the "current cell". This is the cell hierarchy shown in the main window. The current cell is acted on by many of the commands in *Xic*, and in particular only the current cell can be modified. The current cell can be set in many ways, including using the **Open** command in the **File Menu**, or the **Cells Listing** panel from the present menu. One can set the current cell to a subcell with the **Push** command. This can be used in conjunction with the **Info** command in the **View Menu** to push to the cell containing a selected object, to any depth in the hierarchy. The **Pop** command can be used to climb back up the hierarchy to the original current cell.

Cell Menu			
Label	**Name**	**Pop-up**	**Function**
Push	`push`	none	Make subcell the current cell
Pop	`pop`	none	Make parent cell the current cell
Symbol Tables	`stabs`	**Symbol Tables**	List of cell symbol tables
Cells List	`cells`	**Cells Listing**	List cells in memory
Show Tree	`tree`	**Cell Hierarchy Tree**	Display cell hierarchy

9.1 The Push Button: Push Editing Context

Pressing the **Push** button in the **Cell Menu** will push the editing context to a subcell. This means that the subcell becomes the "current cell", and editing operations can be performed in this cell. The **Pop** command in the **Cell Menu** can be used to return to the original current cell.

If, when the **Push** button is pressed, the **Info** command is active and an object is selected that is not in the current cell, The editing context will be pushed to the cell containing that object, which may be arbitrarily deep in the hierarchy.

Otherwise, if any subcells are selected, the editing context will be pushed to the most recently selected subcell. If no subcell has been selected, the user is asked to select one.

The pushed-to cell is displayed in true orientation, with or without the surrounding context shown as set with the **Show Context in Push** button in the **Main Window** sub-menu in the **Attributes Menu** or in the sub-window **Attributes** menu. The surrounding context is generally shown with reduced illumination to visually differentiate the current cell from the context. The illumination percentage can be set in the **Window Attributes** panel (from the **Attributes Menu**), or equivalently by setting the ContextDarkPcnt variable to a value 1-100 (100 indicates no darkening).

The history of which cells have been pushed to and popped from is saved. Assume that previously one has pushed into the hierarchy and popped back. When the **Push** button is active, pressing the **Enter** key will push down one level and deactivate the button. Holding the **Ctrl** key while pressing **Enter** will suppress the button deactivation, so that one can press **Enter** repeatedly to push deeper into the hierarchy, following the last push sequence. Pressing **Shift-Enter** will cycle backwards, i.e., pop, with button deactivation controlled by the **Ctrl** key as above. Unless the **Ctrl** key was up during the last context change, the **Push** command is still active and one must press **Esc** before the cell can be edited.

If instead of pressing **Enter** a subcell is clicked on, the subcell is pushed to in the usual way, and all past history below the present level is removed.

9.2 The Pop Button: Pop Context

Pressing the **Pop** button in the **Cell Menu** will pop the editing context back to the parent cell, if the **Push** command has been employed.

If the user switches between physical and electrical mode while a push is active, the symbol currently being edited remains the target, but the cell becomes top-level (not in a push) in the new mode. If the original mode is returned to without editing a different cell, the push stack is retained. If a new cell is edited in the new mode, through a push or otherwise, the original push context is lost. This context is also lost if the **Clear** function in the **Cells Listing** from the **Cell Menu** is invoked.

9.3 The Symbol Tables Button: Switch Symbol Table

The **Symbol Tables** panel is brought up with the **Symbol Tables** button in the **Cell Menu**. A "symbol" is a cell name, which applies to corresponding physical and electrical cells. A symbol table is a container (a hash table) which holds cell definitions in memory for rapid access by name. Within a symbol table, all cells have unique names, and an attempt to add a cell with an existing name will simply overwrite the existing cell in the table. On program startup, a default symbol table is provided, which will contain all cells unless the user intervenes.

It is possible to have multiple symbol tables available. This allows different versions of a cell with the same name to exist in memory concurrently, though in different symbol tables. It also provides a means for the user to "start fresh" without actually destroying cells in memory.

This pop-up manages the symbol tables that are currently allocated. It is possible to add or delete symbol tables, and to switch between the tables. The table in use contains the cell "memory" that is currently available.

The option menu to the left provides the means for switching between existing tables. Each table has a name, which is listed in the menu. Initially, only one table, named "`main` is available.

The **Add** button allows a new symbol table to be created and added to the list. The user is asked to provide a name for the table. This name can be just about any text string, however if the name already exists in the table list, a new table is not created. The table corresponding to the name becomes the current table. Although non-alphanumeric characters can be included in the name, this will require that the name be double-quoted if used in the extended layer name syntax of layer expressions or the **!layer** command.

The **Clear** button will clear and destroy the contents of the current table. After confirmation, if there are modified cells, the user will be given a chance to save them to disk. If the user does not abort, all cells in the current table will be destroyed, and the table will be empty except for the default "`noname`" cell which will be read from disk if it exists, and this will become the current cell.

The **Destroy** button will destroy the current table, and its contents. It is not possible to destroy the "`main`" table, the button is disabled when that table is current. After confirmation, if there are modified cells, the user will be given a chance to save them to disk. If the user does not abort, all cells in the table, and the table itself, will be destroyed. After the table is destroyed, one of the remaining tables will become the new current table.

Note that when switching between tables, the current cell in use at the time of the switch is saved, and recalled when the user returns to that table.

9.4 The Cells List Button: Cell Listing Panel

The **Cells List** button in the **Cell Menu** is used to bring up the **Cells Listing** panel, providing a listing of cell names. The cells listed are dependent upon the context, as will be described, and can be filtered for various criteria. The panel can be used to select cells for editing or placement, among other useful features.

If the **Display** button in the **Cell Hierarchy Digests** panel is active, i.e., the program is in hierarchy display mode, the cells listing is obtained from the CHD currently being displayed. In this case, filtering (to be described) does not apply. Otherwise, the listing is obtained from the cells presently in memory, in the current symbol table.

To the right of the **Dismiss** button is a drop-down menu which provides a choice or electrical or physical display mode for the cells list. The initial selection will be the same as the current display mode. The cells listed will have been created in the selected mode.

The display of the cell names is paged. The number of entries displayed per page can be set with the ListPageEntries variable, or defaults to 5000 if this variable is unset (variables can be set with the **!set** command). If the listing requires multiple pages, a page selection menu will appear to the left of the **Dismiss** button.

Cell names are listed in columns. The top level cells (those that are not used as subcells of another cell) are shown with an asterisk '*', and a plus sign '+' appears for modified cells.

The listing is a drag source, cell names can be dragged and dropped into drawing windows, to display or edit that cell.

9.4.1 Cells Listing Command Buttons

A cell name can be selected by clicking on the name. Only one name can be selected at once, and it will be highlighted.

A number of buttons appear along the left edge of the panel. Without a selection, these buttons are grayed. Selected names are acted on by buttons of the panel, which become active when a selection is made. The buttons enable functionality described below.

Clear

The **Clear** button is available when listing cells from memory, but not in CHD display mode.

This button will clear top-level cells (those not used as a subcell by any other cell in memory, and marked with an asterisk in the list) or all cells from memory. If a top-level cell is selected in the text area, that cell and its descendents which are not referenced outside of the hierarchy are removed from memory, after confirmation. There is no "undo" of this operation. If the cell is not top-level in both electrical and physical modes, the command exits with a warning message. If no cell is selected, the entire symbol table will be cleared (after confirmation). The user is first given a chance to save any unsaved work. The current editing cell becomes the next cell given on the command line, or the default "`noname`" cell if no other cell was specified. This command can not be undone, and anything cleared is very definitely gone.

Tree

The **Tree** button is available in normal and CGD display modes, and is active when a cell name is selected.

The **Tree** button is used to bring up the **Cell Hierarchy Tree** pop-up, which can also be initiated with the **Show Tree** button in the **Cell Menu** (for the current cell). From the **Tree** button in the **Cells Listing** panel, the **Cell Hierarchy Tree** pop-up will display the hierarchy of the selected cell.

Open

The **Open** button is available when listing cells from memory, but not in CHD display mode. The button is active when a cell name is selected.

Pressing the **Open** button will load the selected cell into the main window, for display or editing. Cells can also be dragged from the listing and dropped into drawing windows, with a similar effect.

Place

The **Place** button is available when listing cells from memory, but not in CHD display mode, and is not available in the *Xiv* feature set. When available, it is active when a cell name is selected.

Pressing the **Place** button will cause the selected cell to become the current master cell, and the **Cell Placement Control** panel will appear. Instances of the master can be created by pressing the **Place** button in the **Cell Placement Control** panel, then clicking on locations in a drawing window.

Copy

The **Copy** button is available when listing cells from memory, but not in CHD display mode, and is not available in the *Xiv* feature set. When available it is active when a cell name is selected.

The **Copy** button allows an existing cell to be duplicated under a new name. The user must explicitly save the copied cell to disk if the new cell is not placed in a hierarchy saved as an archive file, otherwise the copied cell will be lost when the program is exited, though the new cell is marked as "modified" so the user will be prompted to save it when exiting. Pressing **Copy** will cause a dialog box to appear asking for a new name for the cell. A copy will be made if the user enters a valid new name, which must not already be in use. The new name will become highlighted in the cell listing.

Any cell can be copied. Copies will always be created with the IMMUTABLE and LIBRARY flags (see below) unset.

Replace

The **Replace** button is available when listing cells from memory, but not in CHD display mode, and is not available in the *Xiv* feature set. When available, it is active when a cell name is selected, and at least one cell instance is selected in a drawing window.

The button allows cell instances selected in a drawing window to be replaced with instances of the selected cell name. Pressing the button brings up a confirmation pop-up. A 'yes' response will initiate the replacement. The current transform is ignored when replacing cells from this panel, which is different from the **Replace** function in the **Cell Placement Control** panel from the side menu.

When a cell is replaced, the placement of the new cell is determined in physical mode by the setting of the **Origin/Lower Left** buttons in the **Cell Placement Control** panel (though it may not be visible). When **Lower Left** is active, the lower left corner of the replacing cell corresponds to the lower left corner of the replaced cell, otherwise the cell's origins are used. In electrical mode, the reference terminal (the first connection point) is always placed at the same location as the reference terminal of the replaced cell.

Rename

The **Rename** button is available when listing cells from memory, but not in CHD display mode, and is not available in the *Xiv* feature set. When available, it is active when a cell name is selected.

The **Rename** button allows a cell in memory to be given a new name. All references to the cell throughout the symbol table will be changed to call the new name. This is useful to avoid name clashes in designs intended to be merged with other designs. Note that the newly named cell should be explicitly saved as a file if in native format, or it may be lost when the user exits. The cell will be saved in the hierarchy if an ancestor cell is written to an archive file. The user must remember to save any cells which call the renamed cell (the MODIFIED flag is set for these cells, so that the user is warned at program exit).

Pressing the **Rename** button brings up a dialog box asking for the new name. The renaming is effective if a valid new name, which must not already be in use, is given.

Leading and trailing white space is stripped from the name, and any non-empty name is accepted, though a warning is issued if the name contains a character that may cause trouble. The GDSII specification allows alpha-numerics plus '`$`' (dollar sign), '_' (underscore), and '`?`' (question mark). A character not in this list will trigger the warning. The user should stick to valid cell names when possible.

Cells with the IMMUTABLE flag (see below) set can not be renamed. Cells with the LIBRARY flag set can be renamed, which will unset the LIBRARY flag.

Search

The **Search** button is available in normal and CGD display modes.

In normal display mode, when the **Search** button is pressed, the listing will initially contain only cells in the hierarchy of the current cell, selections in the listing are ignored. If the user clicks in a drawing window displaying the current cell, the listing will then contain only cells with instances that appear under the click location. If the user drags button 1 to define a rectangle in a drawing window displaying the current cell, only cells that have instances that appear in the drag rectangle will be listed. These operations can be repeated, the listing will be updated after each operation. Pressing the **Search** button again to deactivate it will revert to listing all cells in the current symbol table.

In CHD display mode, when the **Search** button is pressed, the listing will contain cells found in the CHD, including and under the cell currently being displayed in the main window. Clicking or dragging in the window will restrict the cell listing as in the normal display mode.

The label at the top of the **Cells Listing** will show the search area coordinates in microns, unless the InfoInternal variable is set, in which case internal units are given.

Flags

The **Flags** button is available in normal mode only.

Cells in the main database have two flags which can be modified by the user. The IMMUTABLE flag indicates that the cell is read-only, and can not be modified or renamed. The LIBRARY flag indicates that the cell was read through the library mechanism. Cells with the LIBRARY flag set are not included when writing output, unless the **Include Library Cells** check box in the **Export Control** panel is active, or equivalently the KeepLibMasters variable is set.

Cells read into the database through the library mechanism will have both the IMMUTABLE and LIBRARY flags set. The panel that appears when the **Flags** button is pressed allows the user to change the flag states, and corresponding cell behavior.

If no cell name is selected, all of the cells listed in the **Cells Listing** will be displayed in the **Set Cell Flags** panel, along with colored indicators of the status of the two flags. If a cell name is selected, only the selected cell will be listed in the **Set Cell Flags** panel upon pressing **Flags**. Clicking on the indicators will toggle the indicators. The indicators can also be set globally with the buttons above the listing. The **Apply** button must be pressed to actually change the flags in the cells.

Cell flags can also be listed and set/unset with the **!setflag** command.

If the IMMUTABLE flag of the current cell is set, user interface editing features are disabled. The **Enable Editing** button in the **Edit Menu** can also be used to set the state of the IMMUTABLE flag of the current cell.

Setting the LIBRARY flag is a means to prevent cell definitions from appearing in the output file when the hierarchy is written. It is occasionally necessary to use this feature to enforce resolution of cells from another source in a subsequent read, perhaps from a different library or another layout.

It is also useful on occasion to create a customized library cell, which will become part of the user's cell collection. In this case, the LIBRARY and IMMUTABLE flags for the library cell would be unset, and the cell modified to the user's needs, and the user's cell hierarchy written to disk. On subsequent reads, the user's version of the cell, which will exist in the file, will satisfy the references, rather than the version from the library.

Another way to accomplish this, perhaps somewhat safer, would be to copy the library cell to a new name (using **Copy**), and reference instances of the copy instead of the library cell. Copies do not have the flags set (unless reset by the user).

Info

The **Info** button is available in normal and CGD display modes.

In normal display mode, the **Info** button produces a pop-up that provides information about subcells and other objects, as from the **Info** button in the **View Menu**. If a cell name has been selected in the listing, the **Cell Hierarchy Tree** pop-up, or in a drawing window, pressing the **Info** button will display a window containing information about the cell. This information includes the size, number of objects and subcells, and cells for which the selected cell is a subcell. If this button is pressed when there is no selected cell name, the info window will also appear, but contain no data. In any case, when the info window is visible, clicking on objects in drawing windows will reload the window with information about the object.

In CHD display mode, information contained in the CHD is shown, for a selected cell or the displayed top-level cell if there are no selections. The information in the CGD is dependent upon the parameters used when the CHD was created.

Show

The **Show** button is available in normal and CGD display modes.

The **Show** button enables a mode where cell instances are highlighted in the main drawing window. If a cell name has been selected in the listing, all instances of the cell will be outlined in the highlighting color. The outlines apply to all instances of the cell, regardless of the level in the hierarchy or expansion status. This facilitates finding instances of a cell in a complex hierarchy. The display will track the currently selected cell name in the listing. If no selection, no highlighting is shown, until a selection is made. Only one cell can be highlighted at once. The number of instances found of the selected cell will be printed in the prompt area.

Filter
This button brings up the **Cell List Filter** panel, with which one can limit the cell list to those with specific attributes. After specifying the filtering criteria, pressing the **Apply** button in the panel will update the listing. The next section describes this panel.

9.4.2 Cell Filtering

The **Cell List Filter** panel appears when the **Filter** button in the **Cells Listing** panel, which is obtained from the **Cell Menu**. This provides criteria that enables a cell to be listed in the **Cells Listing**.

Each entry contains two check boxes, with logic such that at most one can be set at a time. Each is assiciated with some assertion about a cell. If the left box is checked, the cell will be listed if the assertion is **not** true. If the right box is checked, the cell will be listed if the assertion is true. If neither is checked, the assertion is not tested.

A cell will be listed if all tests indicate that the cell should be listed. If no tests are done, the cell will be listed by default. The available tests are described below.

Immutable
List cells with the IMMUTABLE flag set.

not Immutable
List cells with the IMMUTABLE flag **not** set.

Via sub-master
List cells that are standard via sub-masters (physical only).

not Via sub-master
List cells that are **not** standard via sub-masters (physical only).

Library
List cells with the LIBRARY flag set.

not Library
List cells with the LIBRARY flag **not** set.

PCell sub-master
List cells that are parameterized cell sub-masters (physical only).

not PCell sub-master
List cells that are **not** parameterized cell sub-masters (physical only).

Device
List cells that are devices (electrical only).

not Device
List cells that are **not** devices (electrical only).

Top level
List cells that are top level (not used as a subcell).

not Top level
List cells that are **not** top level.

Modified
List cells that are modified, i.e., have been changed in some way.

not Modified
List cells that are **not** modified.

With alt
List cells that have an alternate-mode cell defined, i.e., in the physical listing, list cells if an electrical mode cell of the same name exists.

not With alt
List cells that have no alternate-mode cell defined.

Reference
List reference cells. These are special cells that reference another cell hierarchy.

not Reference
List cells that are **not** reference cells.

Parent cells
This makes use of the text entry area on the same line which can contain a list of cell names. List cells that use at least one of the listed cells as subcells. If the text entry is empty, list cells that contain subcells.

not Parent cells List cells that do not contain any of the cells listed in the text area as subcells, or list cells that contain no subcells if the text area is empty.

Subcell
This makes use of the text entry area on the same line which can contain a list of cell names. List cells that are subcells of any of the listed cells. If the text area is empty, list cells that are used as a subcell of another cell in memory.

not Subcell
List cells that are not a subcell of any of the cells listed in the text area. If the text area is empty, list cells that are not used as a subcell.

With layers
This makes use of the text entry area on the same line which can contain a list of layer names. List cells that contain objects on any of the layers listed. If the text area is empty, list cells that contain any geometry.

not With layers
List cells that do not have any geometry on the listed layers. If the text area is empty, list cells that have no geometry.

With flags
This makes use of the text entry area on the same line which can contain a list of flag names (see 9.4.3). At least one flag must be given or the test is ignored. List cells that have one or more of the listed flags set.

not With flags
The text area must have at least one entry or the test is ignored. List cells that do not have any of the listed flags set.

From filetypes
This makes use of the text entry area on the same line which can contain a list of file type names from among "`none`", "`native`", "`cgx`", "`cif`", "`gds`", "`oasis`", and "`openaccess`". Only the first two letters are needed. List cells that were read from one of the listed file types. Internally generated cells will have type "`none`". If the list of types is empty, the test is ignored.

not From filetypes
List cells that were not read from the listed file types. The test is ignored if the type list is empty.

When the **Apply** button is pressed, the cell listing in the **Cells Listing** panel will be updated to reflect the given filtering criteria.

The filtering state can be saved to and recalled from five registers, through the **Store** and **Recall** menus. There are separate register sets for electrical and physical display modes.

The filter state can also be expressed as a string, using keywords. Presently, this is used only by the `ListCellsInMem` script function. Each keyword or keyword/value pair represents a clause, and the displayed cells are the logical AND of the clauses given. The available clauses are described below.

`immutable`
List cells with the IMMUTABLE flag set.

`notimmutable`
List cells the IMMUTABLE flag **not** set.

`viasubm`
List cells that are standard via sub-masters (physical only).

`notviasubm`
List cells that are **not** standard via sub-masters (physical only).

`library`
List cells with the LIBRARY flag set.

`notlibrary`
List cells with the LIBRARY flag **not** set.

`pcellsubm`
List cells that are parameterized cell sub-masters (physical only).

`notpcellsubm`
List cells that are **not** parameterized cell sub-masters (physical only).

`device`
List device cells.

`notdevice`
List cells that are **not** device cells.

`toplev`
List cells that are not used as a subcell, i.e., top-level cells.

`nottoplev`
List cells that are used as a subcell, i.e., **not** top-level.

`modified`
List cells with the MODIFIED flag set.

`notmodified`
List cells with the MODIFIED flag **not** set.

`withalt`
List cells that have an alternate-mode cell defined, i.e., in the physical listing, keep cells if an electrical mode cell of the same name exists.

`notwithalt`
List cells without an alternate-mode cell defined.

`reference`
List reference cells.

`notreference`
List cells that are **not** reference cells.

`parent` "*cellname1 cellname2* ..."
This keyword requires a following quoted list of cell names. List cells that use at least one of the cells in the list as subcells. If the cell list is empty, specified by two quote marks `""`, list cells that have subcells.

`notparent` "*cellname1 cellname2* ..."
This keyword requires a following quoted list of cell names. List cells that do not have any of the listed cells as subcells. If the cell list is empty, specified by two quote marks `""`, list cells that have no subcells.

`subcell` "*cellname1 cellname2* ..."
This keyword requires a following quoted list of cell names. List cells that are used as a subcell in one or more of the listed cells. If the cell list is empty, specified by two quote marks `""`, list cells used as a subcell (same as `nottoplev`)

`nosubcell` "*cellname1 cellname2* ..."
This keyword requires a following quoted list of cell names. List cells that are not used as a subcell in any of the listed cells. If the cell list is empty, specified by two quote marks `""`, list cells that are not used as a subcell (same as `toplev`).

`layer` "*layername1 layername2* ..."
This keyword requires a following quoted list of layer names. List cells that have objects on one or more of the listed layers. If the layer list is empty, specified by two quote marks `""`, list cells that have some geometry on any layer.

`notlayer` "*layername1 layername2* ..."
This keyword requires a following quoted list of layer names. List cells that do not have geometry on any of the listed layers. If the layer list is empty, specified by two quote marks `""`, list cells that have no geometry.

`flag` "*flagname1 flagname2* ..."
: This keyword requires a following quoted list of flag names (see 9.4.3). List cells that have at least one of the listed flags set. If the list is empty, the clause is ignored.

`notflag` "*flagname1 flagname2* ..."
: This keyword requires a following quoted list of flag names. List cells that have none one of the listed flags set. If the list is empty, the clause is ignored.

`ftype` "*filetype1 filetype2* ..."
: This keyword requires a following quoted list of file types, from "`none`", "`native`", "`gds`", "`cgx`", "`oasis`", "`cif`", and "`openaccess`". Only the first two letters of the type names are necessary. List cells that were read from one of the listed file types. Internally generated cells will have type "`none`". If the list is empty, the clause is ignored.

`notftype` "*filetype1 filetype2* ..."
: This keyword requires a following quoted list of file types, as above. List cells that were read from a file type that is not in the list. If the list is empty, the clause is ignored.

Examples:

```
notlibrary layer "M1 M2" parent cell1 notparent cell2
```

List cells that are not library cells and that contain objects on `M1` or `M2`, and contain `cell1` but don't contain `cell2`.

```
subcell maincell layer BASE notlayer VIA notparent ""
```

List subcells of `maincell` that have objects on layer `BASE` but have no objects on layer `VIA` and that have no subcells.

9.4.3 Cell Flags

Cells in memory contain a number of flags. Most of these are used internally and can not be set by the user. All set flags can be seen in the **Info** windows when cell data are shown.

The table below lists all flags, with a brief description.

Flag Name	User Set	Set When, or Description
`BBVALID`	N	Cell bounding box is valid
`BBSUBNG`	N	A subcell has unknown bounding box
`ELECTR`	N	Cell contains electrical data
`SYMBOLIC`	N	Cell has active symbolic representation
`CONNECT`	N	Connectivity info is current
`GPINV`	N	Inverted ground plane current
`DSEXT`	N	Devices and subcircuits extracted
`DUALS`	N	Physical/electrical duality established
`UNREAD`	N	Created to satisfy unsatisfied reference
`COMPRESSED`	N	Save hierarchy in compressed form
`SAVNTV`	N	Save in native format before exit
`ALTERED`	N	Cell data were altered when read
`CHDREF`	N	Cell is a reference
`DEVICE`	N	Cell represents a device symbol
`LIBRARY`	Y	Cell is from a user library
`IMMUTABLE`	Y	Cell is read-only
`OPAQUE`	Y	Cell content is ignored in extraction
`CONNECTOR`	Y	Cell is a connector
`SPCONNECT`	Y	SPICE connectivity info is current
`USER0`	Y	User flag 0
`USER1`	Y	User flag 1
`PCELL`	N	Cell is a PCell sub- or super-master
`PCSUPR`	N	Cell is a PCell super-master
`PCOA`	N	Cell is a PCell sub-master from OpenAccess
`PCKEEP`	N	PCell sub-master read from file
`STDVIA`	N	Cell is a standard via sub-master

The flags with a Y in the second column can be set by the user, with the `SetCellFlag` script function and in other places, depending on the flag.

The first two user-modifiable flags are normally controlled by *Xic*, however it is possible for the user to change their state through the **Flags** button in the **Cells Listing** panel, and through the `SetCellFlag` script function.

`LIBRARY`
: This flag is set for cells that were read into memory through the library (see 8.12) mechanism. By default, these cells are not included when a hierarchy is written to disk.

`IMMUTABLE`
: This indicates that the cell is read-only and can't be edited. This will be set for cells read into memory through the library mechanism.

The remaining flags are completely under control of the user, they are not set by *Xic*. These are set via the properties mechanism, from the **Cell Property Editor** (**Flags** property) or with the `SetCellFlag` script function. Using a property to control these flags provides persistence when saved to disk.

`OPAQUE`
: The physical contents of the cell should be ignored in extraction.

`CONNECTOR`
: The cell is a via or other connector that contains no devices.

`USER0`, `USER1`
Convenience flags for the user. *Xic* does not use these, but they may be useful in some application.

9.5 The Show Tree Button: Show Cell Hierarchy

The **Show Tree** button in the **Cell Menu** brings up the **Cell Hierarchy Tree** window, which presents a tree diagram representing cell hierarchy. Each subcell is initially shown unexpanded, but these can be expanded by clicking on the expander symbol. Subcells can be unexpanded by clicking again in the same location. The glyph used to represent the expander is dependent on the GTK theme in use, and may take different forms. Clicking elsewhere in the line will select the subcell name, for use by the **Info**, **Open**, and **Place** buttons.

When the main drawing window is in CHD display mode, meaning that the **Display** button in the **Cell Hierarchy Digests** panel is engaged, the **Cell Hierarchy Tree** will display cells from the displayed CHD, rooted at the default cell of the CHD. Otherwise, the listing represents cells in memory, rooted at the current cell. The **Tree** button in the **Cells Listing** panel can also be used to display the **Cell Hierarchy Tree**, rooted at other cells in memory or in the displayed CHD.

Pressing the **Info** button will display information about the selected cell. In CHD display mode, this is information stored in the CHD when the CHD was created. In normal mode, this is the same **Info** window available in the **View Menu**. Initially, this window will contain information about the selected cell, though subsequent clicks in a drawing window will generate info about other objects.

The **Open** button is only available in normal display mode. Pressing **Open** will open the selected cell in the main drawing window, and make it the current cell for editing and selections.

The **Place** button, also available in normal display mode only, will pop up the **Cell Placement Control** panel, loaded with the selected cell. This enables instantiation of the cell. The **Place** button is not available in the *Xiv* feature set.

Pressing the **Update** button will rebuild the tree internally and redisplay. The tree does not automatically track changes in the cell hierarchy due to editing, the **Update** button can be used to update the tree manually if needed.

The label at the bottom of the panel provides an indication of the complexity of the tree. The total "nodes" would be the number of lines in the display if all items were expanded. The depth is the maximum hierarchy depth found.

The listing is a drag source. Cell names can be dragged and dropped into drawing windows, to display or edit that cell.

This page intentionally left blank.

Chapter 10

The Edit Menu: Edit Layout

The **Edit Menu** contains commands which control aspects of layout editing, such as transformations and other settings, and commands that bring up panels that control cell placement and flattening, property editing, and other functions.

The table below summarizes the commands that appear in the **Edit Menu**, including the internal command name and the command function.

Edit Menu			
Label	**Name**	**Pop-up**	**Function**
Enable Editing	`cedit`	none	Enable/disable editing mode for current cell
Setup	`edset`	**Editing Setup**	Show **Editing Setup** panel
PCell Control	`pcctl`	**PCell Control**	Set pcell options
Create Cell	`crcel`	none	Create new cell
Create Via	`crvia`	none	Create a standard via
Flatten	`flatn`	**Flatten Hierarchy**	Flatten hierarchy
Join/Split	`join`	**Join or Split Objects**	Control join/split operations
Layer Expression	`lexpr`	**Evaluate Layer Expression**	Control layer expression evaluation
Properties	`prpty`	**Property Editor**	Edit properties
Cell Properties	`cprop`	**Cell Property Editor**	Edit cell properties

10.1 Cell, Instance, and Object Properties

A property consists of an integer and a corresponding text string. Every database object, including cells, instances, and geometrical objects, has the native ability to accept properties, though this is enabled selectively. Properties are saved in the design data file along with the item to which it is attached.

The **Property Editor**, which is brought up with the **Properties** button in the **Edit Menu**, provides the primary means of property manipulation of objects found in the current cell. The **Cell Property Editor**, which is obtained with the **Cell Properties** button in the same menu, provides the primary means for manipulating properties of the current cell itself.

Properties can be applied to physical objects and cells by the user, using the user's property number and format, to suit tye user's purposes. This is fine, as long as the user's property numbers are outside of the range reserved by *Xic*. Other properties are set by *Xic* for internal use such as to store the grid used for the layout, or the GDSII end style for wires. Still others may be set by the user, but have significance to *Xic*. In schematic layouts, these properties define electrical parameters for devices, and are used when generating SPICE output. In electrical mode, the user is restricted to a small set of properties understood by *Xic*, whereas in physical mode any non-restricted number and string are allowed. Finally, *Xic* supports "pseudo-properties" which are not actual properties, i.e., they are not stored, but their application modifies or returns some parameter related to the object. These are listed in the **Property Editor** when in physical mode.

10.1.1 Physical Mode Properties

In physical mode, the user has wide freedom to apply properties to cells, subcells, and objects. The only constraint is that the following number ranges are restricted and must be avoided, for arbitrary user-specified properties. However, there are some properties within the range that can be set by the user, that have specific syntax requirements and meaning to *Xic*. These will be described.

The property number ranges used by *Xic* are:

1 – 30
: These numbers are used or reserved in electrical mode only, they can be used freely in physical mode cells, instances, and objects.

7000 – 7199
: Property values in this range are reserved for use by *Xic*, and should (in general) not be assigned by the user.

7200 – 7299
: These values are reserved for the pseudo-properties (see 10.1.2) and can not be used for other purposes.

The **Property Editor** from the **Properties** button in the **Edit Menu** is used to assign properties to cell instances and objects within the current cell. The **Add** button in the editor allows addition of special properties used by *Xic*, and arbitrary user-specified properties. See the **Property Editor** description in 10.10) for a description of the properties that can be added.

The object's existing properties, and available pseudo-properties are listed in the **Property Editor**, with color and syntax coding to indicate the classification. Pseudo-properties are not actual properties, but when applied to an object will change or report some parameter of the object.

The **Cell Property Editor** from the **Cell Properties** button in the **Edit Menu** allows editing of the properties of the current cell. See the **Cell Property Editor** description for a list and discussion of the cell properties that can be added with the **Add** menu.

A description of the physical properties used by *Xic*, including the property string syntax, is provided in D.1.

10.1.2 Pseudo-Properties

Xic supports "pseudo-properties" which when applied are not saved as properties, but rather change or return some parameter related to the object. This allows the property setting mechanism to be used to alter the physical layout, which can be an important feature in design automation.

In physical mode, when the **Property Editor** is in use, the listing will include the available pseudo-properties for the current object. The pseudo-properties can be "added" or "edited" to modify the current object (or all objects if in global mode). As usual, such changes can be undone/redone with the standard operations.

Internally, pseudo-properties can be applied to any object, electrical or physical. Many of the script functions that modify objects use the pseudo-property mechanism internally. These functions can take electrical or physical input. The graphical user interface, though, allows pseudo-properties to be applied in physical mode only, through the **Property Editor**.

The pseudo-properties are listed below, giving the property number and an internal name for the pseudo-property.

7200: XprpType
: This value can be read from all objects. The returned property string consists of a single character: `b`, `p`, `w`, `l`, or `c` for boxes, polygons, wires, labels, or subcells respectively. The returned value indicates the type of object.

7201: XprpBB
: This value can be read from all objects, and can be applied to boxes, polygons, wires, and labels. The property string is in the form *left*,*bottom right*,*top* where the *left*, etc. are the coordinates of the object's bounding box in internal units. The `x` and `y` values are separated by commas. When this property is applied to an object other than a subcell, the object's geometry is stretched to conform to the bounding box given.

7202: XprpLayer
: This value can be read from all objects, and can be applied to boxes, polygons, wires, and labels. The property string is the name of the layer on which the object is defined. For subcells, the returned name is "`$$`", which is the internal name for the layer on which subcells are defined. When this property is given to an object (not a subcell), and if the name is found in the layer table, the object will be moved to the given layer.

7203: XprpFlags
: This value can be read from all objects, and can be applied to all objects. The property string is a list of values and keywords corresponding to special flags associated with the object. These flags are set internally, and should not be set by the general user.

7204: XprpState
: This value can be read from all objects, and can be applied to all objects. The property string contains one of the keywords `normal`, `selected`, `deleted`, `incomplete`, and `internal`. This indicates a state value for the object which is used internally. These values should not be set by the general user.

7205: XprpGroup
: This value can be read from all objects, and can be applied to all objects. The property string is an integer corresponding to the conductor group assigned to the object by the extraction system. Though all objects have this data field, it has relevance to objects that are defined on conducting layers only. It is generally unwise for the user to set this value.

7206: XprpCoords
: This value can be read from all objects, and can be applied to boxes, polygons, wires, and labels. The property string is a list of coordinates, one for each vertex, with the `x` and `y` values separated by a comma. Line feeds are included in returned strings to keep the line length below 80 characters. The values are in internal units. For boxes, labels, and subcells, the coordinates are those of the

bounding box. For polygons and wires, the coordinates are the actual vertices. For all but wires, the first and last coordinates are the same, i.e., the path is closed. For boxes and polygons, applying this property will change the object geometry. If the new geometry is a Manhattan rectangle, the new figure will be a box, otherwise it will be a polygon. When applied to wires, the new object will always be a wire, but with the new path. The coordinates given to a label must describe a Manhattan rectangle, and the label will be stretched to fill the given rectangle, as with applying XprpBB.

7207: XprpMagn

This value can be read from all objects, and can be applied to all objects. The return value is "1.000000" for objects other than cell instances, and the magnification value for cell instances. When applied to an object or cell instance, the size of the object will change, and a "reference point" of the object will remain in a fixed location.

object	**reference point**
box	lower-left corner
polygon	first vertex in internal list
wire	first vertex in internal list
label	label reference point
instance	transformed master origin

7208: XprpWwidth

This value can be read from wires, and can be applied to wires. The property string is the width of the wire in internal units. When applied to a wire, the width will take the new value. This has no effect when applied to objects other than wires.

7209: XprpWstyle

This value can be read from and applied to wires. When the property string is read, only the first character is significant, the rest if any are ignored. This is used to set the end style of the wire to one of three possible states: **flush**, **rounded**, or **extended**. In *Xic*, both wire ends will have the same style. If **flush**, the wire is truncated normal to the edges at the end vertices. If **rounded**, the wire continues beyond the vertex by half the wire width, but is given a rounded (ideally semicircular) shape (this style is rarely used and is not recommended). The **extended** style is similar in that the wire extends a half-width past the vertices, but the end is square. This is the *Xic* default.

leading character	**style**
`f`,`F`, or `0` (zero)	flush
`r`,`R`, or `1` (one)	rounded
`e`,`E`, or `2`	extended

Applying this property to a wire will cause that wire to be rendered with the given end style. The property has no effect if given to objects other than wires.

7210: XprpText

This value can be read from labels, and can be applied to labels. The return value is the text of the label. The full text including encoded hypertext entries is provided. When applied to a label, the label takes the new text. There is no effect if this property is applied to objects other than labels.

7211: XprpXform

This value can be read from and applied to text labels. It controls a set of flags associated with the label which define the presentation attributes.

The general syntax for the string value is

[+|-] [0x]*hex* | *word*[,...]

Optionally, the string begins with a + or - character. If + appears, it indicates that the flag bits that are specified will be set, and those not specified will be unchanged. If - is given, the flag bits specified will be unset, those not specified will be unchanged. If neither, the flags will be set to a new value consisting of the bits specified which are set, other bits are not set.

The remaining part of the string effectively specifies a set of flag bits. This consists of space or comma-separated keywords or hex integers. Hex integers can have an optional "0x" or "0X" prefix. The overall value is the OR of all terms given. The table below lists the accepted keywords and the equivalent flag bits. Keyword recognition is case-insensitive.

Word	Hex Bits	Description
R0	0	no rotation (dummy token)
R45	10	45 degree rotation
R90	1	90 degree rotation
R135	11	135 degree rotation
R180	2	180 degree rotation
R225	12	225 degree rotation
R270	3	270 degree rotation
R315	13	315 degree rotation
MY	4	mirror Y after rotation
MX	8	mirror X after rotation and mirror Y
HJL	0	left justify (dummy token)
HJC	20	center X justify
HJR	40	right justify
VJB	0	bottom justify (dummy token)
VJC	80	center Y justify
VJT	100	top justify
T0	0	text font 0 (dummy)
T1	200	text font 1 (unused)
T2	400	text font 2 (unused)
T3	600	text font 3 (unused)
SHOW	1000	show hidden label
HIDE	2000	hide label
TLEN	4000	show in top-level only
LIML	8000	limit lines

The HJR will override HJC if both are given, similarly VJT will override VJC.

The SHOW/HIDE bits are for implementing a clickable text display, where the label text can be shown or "hidden" by rendering a small glyph instead. At most one of these bits should be set. Either bit overrides the default which is in force when neither is set. These can be applied to any label, however the "clickability" of the label is set by the LabelHiddenMode variable. All labels are "clickable" by default, press **Shift** and click on the label to toggle the hidden/viewing status.

The TLEV bit gives the label the property of being invisible in instances of the containing cell, but visible when the cell is viewed as the top-level (current cell).

The LIML bit causes the label to limit the number of lines displayed, when the label text has multiple lines. The maximum line count defaults to 5, and is otherwise given with the LabelMaxLines variable.

The `TLEV` and `LIML` bits may be applied when reading schematic cells through OpenAccess for Virtuoso compatibility, but are not otherwise used in *Xic*, except as controlled through this pseudo-property.

When applied to a label, the label will be rendered using the new flags. This property has no effect when applied to objects other than labels.

7212: XprpArray
This value can be read from subcell instances, and can be applied to subcell instances. The property string is of the form "*nx*,*ny* *dx*,*dy*" where *nx* and *ny* are the number of columns and rows, and the *dx* and *dy* are the center to center spacings in internal units, for an array of subcells. When applied to an instance, the array parameters of the instance are correspondingly changed. This property has no effect on objects other than subcells.

7213: XprpTransf
This value can be read from subcell instances, and can be applied to subcell instances. The property string is the CIF transformation string for the instance, with coordinates in internal units. When applied to an instance, the instance placement and orientation change to reflect the new transformation. This property has no effect on objects other than subcells.

7214: XprpName
This value can be read from subcell instances, and can be applied to subcell instances. The property string is the name of the instantiated cell. If this property is set, the instance is replaced by an instance of the given cell name. The current transform is added to the existing transform when the new instance is placed. This property has no effect on objects other than subcells.

7215: XprpXY
This pseudo-property has a value that is an x,y coordinate, and can be read from or applied to any object or subcell. The interpretation of this coordinate depends on the type of object. For boxes, it is the lower-left corner. For polygons and wires, it is the first vertex in the vertex list. For labels, it is the text anchor point, and for subcells it is the placement coordinate. Setting the property is equivalent to moving the object.

7216: XprpWidth
This pseudo-property returns the width of any object or cell instance in internal units. It can be applied to objects but not cell instances, and will scale the object to the specified width.

7217: XprpHeight
This pseudo-property returns the height of any object or cell instance in internal units. It can be applied to objects but not cell instances, and will scale the object to the specified height.

The settable pseudo-properties for an object are listed in the **Property Editor**, along with the "real" properties. These can be changed in the same way, which will produce physical changes to the object.

10.1.3 Electrical Mode Properties

In electrical mode, only properties with certain values and data can be entered, and only to objects corresponding to library devices or subcircuit instances.

These properties are generally applied with the **Property Editor** from the **Properties** button in the **Edit Menu**. However, the most frequently used properties, such as those that set device parameters, have values that are shown on-screen as text labels. If one edits the label, the underlying property value is changed as well. This is generally more convenient than using the **Property Editor**. Labels are edited by first selecting a label by clicking on it, then entering the **label** command by pressing the button in the side menu.

A description of the electrical properties used by *Xic*, including the property string syntax, is provided in D.2.

See the description of the **Property Editor** for a listing and discussion of the properties that can be added with the **Add** menu.

Properties of the current call can be added or modified with the **Cell Property Editor** from the **Cell Properties** button in the **Edit Menu**, and are listed in the **Add** menu of the editor. See the **Cell Property Editor** description for a list and discussion of these properties.

10.2 The Enable Editing Button: Enable Cell Editing

This button tracks the state of the IMMUTABLE flag of the current cell, and will alter the flag if editing mode is changed. When the current cell is IMMUTABLE, it can not be modified, and all editing features are disabled. The side menu is hidden, the **Modify Menu** is disabled, and all but this button and **Create Cell** are disabled in the **Edit Menu**.

If there is no current cell, editing features are disabled.

Cells read into *Xic* through the library mechanism have the IMMUTABLE flag set. This button can be used to allow modification of these cells.

Setting the IMMUTABLE flag of the current cell from the **Flags** button in the **Cells Listing** panel from the **Cells Menu** or with the **!setflag** command will have the same effect as use of this button.

10.3 The Setup Button: Show Editing Setup Panel

The **Setup** button in the **Edit Menu** brings up the **Editing Setup** panel. The panel provides controls for setting various parameters and options which apply during layout editing.

Constrain angles to 45 degree multiples

When this check box is active, wire and polygon vertices are constrained to form angles of multiples of 45 degrees. By default, a “smart” path generator is employed, which will construct a valid path to the pointer location from the previous point during wire or polygon construction. This will often add two vertices: a 45 degree extension, followed by a Manhattan extension, in order to connect the points. If the **Ctrl** key is held while the new point is defined, the “smart” feature is disabled, and only one new vertex is added. If the **Shift** key is held, then the 45 degree constraint is removed entirely.

The Constrain45 variable tracks the state of this check box, and setting or clearing the variable will also set or clear the mode.

When active, rotation angles available in the **spin** command, and translation angles in the **Stretch** command, and the vertex editors for polygons and wires, are constrained to multiples of 45 degrees. However, pressing the **Shift** key will remove the constraint in these commands while the key is held. If the Constrain 45 variable is not defined, holding **Shift** will impose the 45 degree angle constraint. Thus, the **Shift** key inverts the effective state of the Constrain 45 variable (and this check box) in these commands.

Merge new boxes and polys with existing boxes/polys

When this check box is set, new boxes and polygons that are created with the side menu commands are merged with existing boxes and polygons to form a larger polygon in the database. New wires will be connected to existing wires on the same layer with the same width and endpoint, but do not participate in the box/polygon merging of new objects.

However, on layers with the `NoMerge` technology file keyword set, merging is always suppressed.

The state of this check box tracks the logically inverted state of the boolean variable NoMergeObjects, which can (equivalently) be set with the **!set** command.

Existing objects can be similarly joined, or split into trapezoids, with the buttons in the **Join or Split Objects** panel brought up with the **Join/Split** button in the **Edit Menu**.

Join (merging) operations are subject to the settings of several variables, which have equivalent entries in the **Join or Split Objects** panel. These limit the complexity of polygons created by merging, mostly for optimizing for speed for merging large object collections.

The **Clip and merge new boxes only, not polys** button in this panel modifies the merging behavior to clip and merge boxes only.

This object merging is separate and unrelated to the box merging available when reading a layout file into the database, which has a separate merging control in the **Setup** page of the **Import Control** panel from the **Convert Menu**.

Clip and merge new boxes only, not polys

When merging is enabled (the **Merge new boxes and polys with existing boxes/polys** check box is active), and this check box is also active, polygons are ignored when merging, and new boxes are clipped/merged. This was the merging behavior in releases prior to 3.1.7.

This setting has no effect if merging is not enabled. It tracks the state of the NoMergePolys variable.

Prompt to save modified native cells

When the box is checked, the user will be prompted to save the current cell if the cell is modified, and would be saved as a native cell file, and a new current cell is about to be set. This was standard behavior in releases earlier than generation 4. Although it is always a good idea to save work periodically, the prompt can be annoying to experienced users and is now disabled by default. The user will be given the chance to save modified cells when exiting *Xic* in any case.

This check box tracks the state of the AskSaveNative variable. The variable can be set as a boolean or cleared to change the mode, which is equivalent to checking or un-checking this check box.

No wire width change in magnification

When the box is checked, the width of wires does not change when the wire undergoes magnification, in a **Move**, **Copy**, or **Flatten** operation.

This check box tracks the state of the NoWireWidthMag variable. The variable can be set as a boolean or cleared to change the mode, which is equivalent to checking or un-checking this check box.

Allow Create Cell to overwrite existing cell

When this check box is active, The **Create Cell** operation in the **Edit Menu** and the `CreateCell` script function can overwrite cells already in memory. This can be dangerous and is prevented by default, and the user is advised to be careful if using this feature.

This check box tracks the state of the CrCellOverwrite variable. The variable can be set as a boolean or cleared to change the mode, which is equivalent to checking or un-checking this check box.

Maximum undo list length

This integer entry sets the number of operations remembered in the **Undo** command. If not set, 25 operations are saved. If set to zero, the length is unlimited.

This entry tracks the value of the UndoListLength variable. The variable can be set as an integer or cleared to change the value, which is equivalent to changing the integer entry in this panel.

Maximum number of ghost-drawn objects

This integer entry sets the maximum number of objects to render individually as "ghosts" attached to the mouse pointer during operations such as move and copy. This can be set to an unsigned integer in the range 50–50000. If there are more than this number, some outlines won't be shown, the smaller-area objects will be skipped. The default is 4000 if this variable is not set. If, when moving a large number of objects, the pointer motion is too sluggish, the user can set this variable to compensate.

This entry tracks the value of the MaxGhostObjects variable. The variable can be set as an integer or cleared to change the value, which is equivalent to changing the integer entry in this panel.

Maximum subcell depth in ghosting

This menu sets the maximum expansion depth for instance expansion in ghosting. If **as expanded**, this is the same as the normal expansion depth. The actual expansion depth used in ghosting will not be larger than the normal expansion depth, but can be smaller. For example, setting this to 0 (zero) will prevent expansion of ghosted subcells entirely.

This entry tracks the value of the MaxGhostDepth variable. The variable can be set as an integer or cleared to change the value, which is equivalent to changing the integer entry in this panel.

10.4 The PCell Control Button: PCell Control Panel

The **PCell Control** button in the **Edit Menu** brings up the **PCell Control** panel. From the panel, various options related to parameterized cells (pcells, see 5.1) can be set. The following elements are available.

Auto-abutment mode

The drop-down menu provides three choices, **Mode 1** through **Mode 3**. This provides the three values for the otherPinsOnNet parameter that is part of the Ciranova auto-abutment protocol as implemented in *Xic* (see 5.5). How the pcell uses this parameter is up to the pcell author, there is really no *a-priori* interpretation.

The Ciranova Nmos2 example pcell interprets the value to have the following meanings. This is likely to be used in other pcells as well.

Mode 1 value 0	Auto-abutment is disabled.
Mode 2 value 1	Abutment takes place with no contact between the gates.
Mode 3 value 2	Abutment takes place with a contact (to layer M1) between the gates.

This setting tracks the value of the PCellAbutMode variable. The default (when the variable is not set) is Mode 2. The variable can be set to the integers 0–2, which correspond to the three modes.

Hide and disable stretch handles

Xic implements the Ciranova stretch handle protocol (see 5.4). By default, the stretch handles are visible in selected, expanded instances, and in the current cell if the cell is a sub-master. Setting this option will hide and disable all stretch handles.

The PCellHideGrips variable tracks the state of this check box.

Instance min. pixel size for stretch handles

The stretch handles of a selected instance are shown only if the instance is displayed large enough to avoid inadvertently engaging the stretch handles when using the mouse for other purposes. By default, the smaller of the instance width or height must be 100 pixels or larger for stretch handles to appear. This numerical entry will modify this threshold.

The PCellGripInstSize variable tracks the value of this entry.

List sub-masters as modified cells

When this check box is checked, the **Modified Cells** panel, which appears when exiting *Xic* or from the **!sa** command and **Save** button (in the **Edit Menu**) when there are cells that are modified and unsaved, will include sub-master cells. Normally, sub-masters live only in memory, to be re-created when needed, and are excluded from the listing. However, there may be times when it is desirable to write these to disk.

The PCellListSubMasters variable tracks the state of this check box.

Show all evaluation warnings

By default, certain warning messages are suppressed while evaluating a pcell script, since they can be annoying and are probably only of interest to the pcell author. At present, this applies only to coincident (duplicate) object checking. If this check box is checked, these messages will pop up in a window when the pcell sub-master is created.

The PCellShowAllWarnings variable tracks the state of this check box.

10.5 The Create Cell Button: Create New Cell

The **Create Cell** button in the **Edit Menu** will create a new cell from the currently selected objects. The user is prompted for a name for the new cell. The new cell is created in memory, and should be saved to disk if future use is intended. It is marked as "modified" so the user will be given the chance to save it when exiting *Xic*.

The **!sqdump** function is similar, but writes a native file to disk and does not create a cell in memory.

In electrical mode, note that the new cell is not a subcircuit. It must be edited and connection points added (with the **subct** command) before it can be used in another circuit.

The user is given the option to replace the selected objects with an instance of the new cell.

By default, an attempt to overwrite a cell already in memory will fail. If the CrCellOverwrite variable is set, existing cells in memory can be overwritten (use this with care).

10.6 The Create Via Button: Create Standard Via Variant

The **Create Via** button in the **Edit Menu** brings up the **Via Creation** panel, if standard vias are defined in the current technology. If no standard vias are defined, the menu entry will be grayed, and the panel will not be available. This will be the case for the example `scmos` and other technology files provided. The `xic_tech.demo` technology file found with the memory chip examples does provide standard via definitions, should the user wish to try this feature.

The panel will also appear if the user clicks on a selected instance of a standard via with the **Ctrl** key held. The instance can be reparented to a master with a different parameter set.

The panel contans a number of entry areas, corresponding to the standard via parameters as described below. Of these, the numerical parameters can be changed by the user to create variants. The fields that contain layer names can not be changed, except by creating a new standard via definition. Presently, this must be done by editing the technology file.

Each row of the panel contains a description and two entry areas, as most of the entries have separate values for X and Y directions. Dimensions are in microns.

Via name, cut layer
This row contains two menus, which together provide access to all of the available standard via definitions. The menu on the right provides the process layer names that are used as “cuts”. These are the layers that represent holes in an insulating layer, generally called “via layers”. The menu on the left provides the names of standard vias defined which use that via layer. Most often, there is only one such definition, for a metal to metal contact. In other cases, one of the “conductors” may be an implanted area, in which case there may be several choices. When the user selects a standard via using these menus, the other fields in the panel will be set to the default values for the various parameters.

Layer 1, Layer 2
These are the layer names of the two layers to be connected by the via. These can not be changed by the user, except by selecting another standard via. In *Xic*, **Layer 1**, and the ‘1’ designation in general, corresponds to the bottom conductor.

Cut width, height
The “cut” is the feature on the via layer that actually forms the contact. In a standard via, this is always rectangular. In a semiconductor process, this is almost always a square of a fixed size. Although the two dimensions can be changed by the user, one must be aware of the relevant design rules before doing so.

Cut rows, columns
In order to lower contact resistance and handle higher current, the cut structure can be arrayed. In some situations it may be possible to simply increase the size of a single cut, but in more advanced processing the cut size is fixed and arrays are used. This is probably the most common variant.

Cut spacing X,Y
This is the space between cut edges (not center-to-center) in the X and Y directions. This is only useful if the cut is arrayed. In general, the two values are the same, and fixed at a minimum from a design rule. The values should be changed only with knowledge of the appropriate design rules.

Enclosure 1 X,Y
In addition to the cut, the via will also contain squares of the two metal layers. The “Enclosure” is the distance the metal layer overhangs the cut. The two numbers apply to the bottom layer, in the X and Y directions. The two numbers are typically set to a design rule minimum, and should be changed only with knowledge of the design rules involved.

Offset 1 X,Y
If the offset parameters are zero, the metal rectangle is centered on the cut. One can set the offset to a nonzero value, which will move the center of the metal rectangle relative to the center of the cut. This entry applies to the bottom conductor. These entries are almost always zero.

Enclosure 2 X,Y
As for **Enclosure 1**, but the values apply to the top conductor.

Offset 2 X,Y
As for **Offset 1**, but the values apply to the top conductor.

Origin offset X,Y
: This is the origin of the sub-master coordinate system, which if zero is centered on the cut array. This is the same as Virtuoso, but appears to differ from the OpenAccess specification which seems to indicate that the origin is centered on the lower-left cut element. All features of the via are drawn relative to this offset, so there are no design rule implications. These values are most often zero.

Implant 1, Implant 2
: Up to two additional rectangles can be drawn in the via, representing implant areas. These may apply when contacting activated substrate areas, where additional spacing rules to an implant region edge may apply. If defined in the standard via definition, one or both of these entries may contain an implant layer name. If not, the entry area is grayed. The layer names can not be changed by the user, except by selecting another standard via.

Implant 1 enc X,Y
: If an **Implant 1** layer is present, these entries will contain the enclosure values of the implant 1 layer rectangle relative to the bottom conductor rectangle. That is, the implant 1 rectangle will overhang the bottom conductor rectangle in the X and Y directions by the values given.

Implant 2 enc X,Y
: If an **Implant 2** layer is present, these entries will contain the enclosure values of the implant 2 layer rectangle relative to the top conductor rectangle. That is, the implant 2 rectangle will overhang the top conductor rectangle in the X and Y directions by the values given.

The easiest way to understand the effect of these parameters is to create some vias.

Entering the parameters has no effect until the **Apply** button is pressed. When **Apply** is pressed, a new internal sub-master cell for the variant is created if necessary, and the via structure is ghost-drawn and attached to the mouse pointer. Instances of the via will be placed where the user clicks in a drawing window. As for normal subcells, the current transform will be applied to the via. Most process rules will not accept 45-degree rotations. The placement mode can be exited by pressing the **Esc** key.

10.7 The Flatten Button: Flatten Hierarchy

The **Flatten** button in the **Edit Menu** brings up a small **Flatten Hierarchy** pop-up which controls flattening of the hierarchy by moving the contents of selected subcells into the current cell. A **Depth** choice menu selects the depth into the hierarchy to flatten. If 0, geometry in the selected subcells is brought into the current cell, and sub-subcells are placed in the current cell, becoming subcells. If "`all`", the entire subcell hierarchy is flattened, i.e., all geometry under a selected subcell is brought into the current cell.

The **Don't flatten standard vias, move to top** check box enables to option to keep standard vias as cell instances rather than converting to geometry. Standard vias are otherwise treated as any other cell instance. This check box tracks the state of the NoFlattenStdVias variable. Applies to physical cells only.

The **Don't flatten param. cells, move to top** check box enables to option to retain parameterized cell (pcell) instances rather than converting to geometry. Parameterized cell instances are otherwise treated as any other cell instance. This check box tracks the state of the NoFlattenPCells variable. Applies to physical cells only.

The **Ignore labels in subcells** check box will skip promoion of labels found in subcells being flattened to new labels in the current cell. This is to avoid assigning labels to flattened wire nets that might be ambiguous when naming the net. This check box tracks the state of the the NoFlattenLabels variable. Applies to physical and electrical cells.

The **Use fast mode** check box will select a processing mode that will skip undo list processing and object merging operations for speed and reduced memory use. This may be desirable for large jobs containing complex cells, which may take a long time to process. In this mode, there is no "undo" capability, however.

If the **Use object merging when flattening** check box is checked, the new object merging will be performed when objects from the subcells are promoted to the current cell. This is the same merging as specified in the **Editing Setup** panel from the **Edit Menu**. Use of full polygon merging can greatly increase processing time, simple box clipping/merging has much lower overhead. Merging will generally reduce the object count in the layout.

The **Flatten** button on the pop-up initiates the operation. The subcells to be flattened must have been selected at this point.

Pressing **Ctrl-c** will pause the process, and give the user the option of terminating the job. It is usually not desirable to stop in the middle of a flatten operation, but invoking this prompt may reassure the user that the operation is in progress and not "hung".

In electrical mode, symbolic instances and library devices are never flattened, they are considered atomic. If you must flatten an instance that is displayed symbolically, the instance must first be forced to display as a schematic, either by reverting its master to non-symbolic temporarily, or by adding a NoSymb property to the instance with the **Property Editor**.

10.8 The Join/Split Button: Join or Split Objects

The **Join/Split** button in the **Edit Menu** brings up the **Join or Split Objects** panel. This panel contains controls for setting defaults and initiating join and split operations. These operations are identical to those available from the **!join** and **!split** text commands.

The panel contains the following controls:

No limits in join operation
This check box unsets the limits on the complexity of polygons that are created during the merge, by setting the JoinMaxPolyVerts, JoinMaxPolyGroup, and JoinMaxPolyQueue variables to "0" (zero).

Maximum vertices in joined polygon
This provides an entry area for setting the value of the JoinMaxPolyVerts variable, which limits the number of vertices allowed in a polygon created as the result of a join operation.

Maximum trapezoids per poly for join
This provides an entry area for setting the value of the JoinMaxPolyGroup variable, which places a limit on the number of connected trapezoids that can be used to form a polygon.

Trapezoid queue size for join
This provides an entry area for setting the value of the JoinMaxPolyQueue variable, which provides a limit on the number of trapezoids that can be considered for joining into polygons in a single pass.

Clean break in join operation limiting
When this check box is set, *Xic* will attempt to break polygons where the vertex limit is reached

into pieces so that the boundaries are more visually attractive. This tracks the state of the JoinBreakClean variable.

Include wires (as polygons) in join/split
If this check box is set, wire objects will be included in join/split operations, treated as polygons. If not checked, wires are ignored in these operations. This tracks the state of the JoinSplitWires variable.

Join
This push button initiates a join operation on selected objects. All suitable selected objects will be joined on their respective layers, as for the **!join** command without an argument. Unlike other commands that join (merge) objects, this overrides the `NoMerge` technology attribute if set on an object's layer.

Join Lyr
This will join objects on the current layer, whether selected or not. The layer must be visible, and not have the `NoMerge` technology attribute set. This is equivalent to the **!join** command with the "layer" argument.

Join All
This push button initiates a join operation on all objects in the current cell, selected or not. It applies to objects on visible, selectable layers that do not have the `NoMerge` technology attribute applied. This is the same as the **!join** command with the "all" argument given.

Split Horiz
This will decompose complex polygons into a collection of trapezoids (boxes and simple polygons in the database) that collectively cover the same area and do not overlap. The splitting is performed using horizontal scan lines. This is the same effect as the **!split** command. Wire objects will also be split if the JoinSplitWires variable or the corresponding check box is set.

Split Vert
This will also decompose complex polygons into a collection of trapezoids, however vertical scan lines are used. This is the same effect as using the "`v`" argument to the **!split** command. Wire objects will also be split if the JoinSplitWires variable or the corresponding check box is set.

10.9 The Layer Expression Button: Evaluate Layer Expression

The **Layer Expression** button in the **Edit Menu** brings up the **Evaluate Layer Expression** panel. Layer expressions are logical expressions referencing the geometry on existing layers, with various operators and functions. These evaluate to a pattern, which can be applied to a new or existing layer. Operations include polarity inversion, intersection, union, and many more complex possibilities.

The **Evaluate Layer Expression** panel allows layer expressions to be applied to the current cell hierarchy, much the same as the text-mode **!layer** command. The panel allows easy setting of variables which control the expression evaluation, whether initiated from the panel or the **!layer** command.

Full layer expression evaluation is available in physical mode only, though joining, splitting and copying are available in either mode.

The controls found in the **Evaluate Layer Expression** panel are described below.

To layer
This entry area requires the name of a layer on which new geometry will be created while the layer

expression is evaluated. This is the only control in the panel that requires an entry. If the layer name does not match an existing layer name (as a short or long name), a new layer is created, with a name generated from the given name in the same way as in the technology file layer definitions.

If no expression is given, the **To layer** is created, if it does not exist. If the layer exists, and one of **Joined**, **Horiz Split**, or **Vert Split** is set, that operation will be performed on the **To layer**. The result is similar to the corresponding operations as initiated form the **Join or Split Objects** panel from the **Join/Split** button in the **Edit Menu**, or the **!join** and **!split** commands. If the **To layer** did not previously exist, or the **Default** new object format is selected, layer creation is the only operation performed.

Depth to process

When the layer expression is evaluated, the layer geometry used in the processing is obtained to this level in the hierarchy. If 0, only the geometry in the present cell is considered. If "`all`", the geometry of the complete hierarchy is taken.

Recursively create in subcells

This check box has effect only if the **Depth** is not zero. When checked, the layer expression is evaluated in the current cell and each subcell to depth, using only the objects from that cell. If this is unchecked, the operation is quite different. In this case, there is no recursion, and all new geometry is created in the current cell, but geometry in cells to depth is considered when creating the new geometry.

Partition Size

To maximize computation speed, the expression evaluation is performed step-wise over a logical grid in the target cell. The grid origin is the lower-left corner of the cell. The partition size is the width of an (assumed square) grid cell. The calculations are performed for each grid square that overlaps the cell area. This can be more efficient than calculating the whole cell in one shot (which might not even be possible due to memory limitations).

The gridding is used only if an actual expression is given, and not simply a layer name (or no expression at all). If the expression consists only of a layer name, processing requires only a simple copy and there would be no reason to use partitioning.

If the **None** button is pressed, no partitioning will be used.

The default partition size is 100 microns, which can be adjusted for best performance. The size should be large enough to minimize the number of grid cells to evaluate, but small enough to limit the amount of geometry to process on average in each grid, to avoid huge memory consumption and other ill effects of taking too big of a "bite".

For simple cells, the grid size can be large, or partitioning can be skipped entirely. Partitioning can be skipped by pressing the **None** button, or by setting the size to a value larger than the cell bounding box width and height.

This entry tracks the state of the **Partition Size** variable, which is also used by the **!layer** command and elsewhere.

Number of helper threads

PRELIMINARY. EXPERIMENTAL!

Multiple threads can be used when evaluating a layer expression over a grid. Evaluation in each of the grid cells can be done in parallel, so these jobs are submitted to the thread pool. This allows processor cores to work simultaneously on different parts of the grid.

Multi-threading will be used if this entry is set nonzero. The value is the number of helper threads that can be called upon to parallelize the operation. The speediest value is probably one less than twice the number of available processor cores, as each Intel core provides two hardware threads.

Your results may differ, so one should experiment. One can also experiment with the partition size to get fastest results, larger partitions are more likely to overcome the multi-threading overhead.

This should not be set to a value larger than the number of available hardware threads minus one, but one might wish to try smaller values. If set to a larger value, software threads will be used, which will increase computation time. If set to 0, the operation is single-threaded.

This entry tracks the value of the Threads variable.

Don't clear layer before evaluation
By default, the layer given in the **To layer** entry is cleared before the expression is evaluated, so that the layer will contain only the result of the operation. It this check box is set, the **To layer** will not be cleared, new data will appear in addition to existing data.

New object format
This consists of four interlocking "radio" buttons which establish the nature of the new objects created by evaluating the layer expression. If **Joined** is selected, objects will be combined into polygons before being added to the cell. If **Horiz Split** is selected, objects are added as trapezoids, with a horizontal orientation (maximal width) favored. If **Vert Split** is selected, objects will also be added as trapezoids, however a vertical orientation (maximal height) is favored.

The **Default** choice has the same effect as **Joined** in cases where the layer expression contains more than a layer name, i.e., it contains at least one operator, function, or numeric entry. If the expression consists of a layer name only, the **Default** choice will read the objects from that layer and add them to the **To layer**, without modification. The other new object format choices will cause the objects read from the layer to be joined or split before being added to the **To layer**.

When joining objects, there are several variables which fine-tune the operation. These are most conveniently set from the **Join or Split Objects** panel brought up by the **Join/Split** button in the **Edit Menu**.

Expression
This entry area contains the layer expression to evaluate. This is an expression consisting of existing layer names, operators, and function calls, which will be evaluated. Dark areas will be rendered on the layer given in the **To layer** entry.

Thus, this provides a means of creating a new layer from geometry on existing layers. Labels are ignored during processing, but all other objects contribute. The same layer name can appear in the **To layer** entry and in the expression, in which case the contents of that layer is updated with the result of the expression.

There are eight registers which can be used to save and recall layer expression strings, for convenience. The **Save** and **Recall** buttons provide access to these registers. Selecting an item in the **Save** menu will save the current contents of the **Expression** entry in that register. Selecting an item in the **Recall** menu will load that text into the **Expression** entry area.

Use object merging while processing
When this check box is set, new objects created during evaluation of the layer expression will be merged with existing objects, using the same object merging as specified in the **Editing Setup** panel from the **Edit Menu**.

If there is no **Expression** given, or the expression consists only of the same layer name given in **To layer**, then merging is not performed.

In every other case, the merging enabled from the **Edit Menu** will be performed as new objects are added to the **To Layer**. This merging will defeat the purpose of the join and split format choices, so one must consider when merging makes sense. Merging applies to objects initially on the **To layer**, if not clearing, plus the accumulated objects added as the operation progresses.

Full polygon merging can greatly increase the time and memory required to process a large job. Box clipping has much less overhead.

Fast mode
When set, undo list processing and object merging will be skipped, which reduces memory use and computational overhead to a minimum. However, the operation can not be undone, so this mode should be used with care.

Evaluate
Pressing this button will create the **To layer** if necessary, evaluate the layer expression, and add the newly created geometry to the current hierarchy.

10.9.1 Examples

Clear layer `M0`
To layer: `M0`
Expression: `0`

Copy layer `M1` to layer `NEW`
To layer: `NEW`
Expression: `M1`

Copy the inverse of layer `M1` to `NEW`
To layer: `NEW`
Expression: `!M1`

Copy the intersection area of `I1` and `I2` to `NEW`
To layer: `NEW`
Expression: `I1&I2`

Copy the `R1` and `R2` areas to `New`
To layer: `NEW`
Expression: `R1|R2`

10.9.2 Extended Layer Names

The layer names in the layer expression (but not the **To layer** entry) can actually be given in an extended form:

lname[`.`*stname*][`.`*cellname*]

Most generally, the "layer" name consists of three tokens, two of which are optional (indicated by square brackets above). The tokens are separated by a period ('`.`') character. The individual tokens can be double-quoted (i.e., using the double-quote ('`"`') character), which must be used if the tokens contain non-alphanumeric characters. The period separators must appear outside the scope of any quoting.

lname
This is a short or long layer name, as found in the layer table.

stname
The name of a symbol table which contains the *cellname*.

cellname
The name of a cell.

If only one separator appears, the token that follows is taken as the *cellname*, and the current symbol table is assumed.

The *cellname* is the name of a cell used as the source for geometry. If no *cellname* is given, the name of the current cell is understood. The odd case of an empty *stname* indicates the "`main`" symbol table, e.g., `layer..cell` is equivalent to `layer.main.cell`.

If the *cellname* starts with the '' character, and no symbol table name is given, then the rest of the *cellname* is taken as the name of a "special" database, as created with script functions like `ChdOpenZdb`. If found, geometry will be obtained from the database rather than a cell. Otherwise, when a *cellname* is given, the geometry is obtained from the given cell, as if it were overlaid on the current cell. The *cellname* (or any of the three tokens) can be double quoted, and must be quoted if the name contains a '`.`' character, for example `CPG."mycell.xic"`.

If a *stname* is given, and the name matches an existing symbol table name, the cell is obtained from that symbol table. If the symbol table name is given, the *cellname* field must appear, but can be empty (a trailing period) which indicates the name of the current cell.

If the *stname* is given, and the cell is not in this table, it will be opened from disk into the given table (not the current table) if found as a native cell file in the search path.

The coordinate origin of the source cell is taken as the origin of the current cell. The source cell must be in memory, or be in a native cell in the search path.

Objects read from a "special" database are clipped to the boundary of the cell being added to. No such clipping is done when objects are read from another cell.

10.9.3 Advanced Examples

Suppose one has two versions of a cell, `cell` and `cell_old`, and one needs to know if they differ on layer `M1`. Open a dummy cell for editing, then supply the following and evaluate.

To layer: `ZZ`
Expression: `M1.cell^M1.cell_old`

Press the **Home** key to view the entire cell space. Any geometry shown on the new dummy layer `ZZ` is the exclusive-OR of the geometry on `M1` of the two cells, i.e., the difference. If there is no geometry on `ZZ`, `M1` is the same in `cell` and `cell_old`.

As a variation, suppose that the user has done the following:

Set symbol table to "`old`".
open `oldstuff/mycell`
return to previous symbol table
open `newstuff/mycell`

There are now two versions of `mycell` in memory. To compare the layer `M1` in the two cells, one could then evaluate

To layer: `ZZ`
Expression: `M1^M1.old.`

Then the `ZZ` layer, which consists of the exclusive-OR of old and new `M1` in `mycell`, would be added to the current `mycell`. Pressing the **Tab** key undoes the addition.

Suppose one wants to import the inverse of the geometry on layer `VIA` from `cell` into the current cell, also on layer `VIA`:

> **To layer**: `VIA`
> **Expression**: `!VIA.cell`

The `VIA` layer now consists of the inverse from `cell`. Any geometry that existed on `VIA` in the current cell before the command was given is deleted (assuming that the **Don't clear** check box is unchecked). The bounding box of the current cell may have been expanded to include the bounding box of `cell`. The area used to create an inversion is the rectangle bounding all cells referenced in the expression, plus the current cell.

Suppose one simply wants to copy the geometry from layer `M2` of `cell` into the current cell:

> **To layer**: `M2`
> **Expression**: `M2.cell`

The `M2` layer now consists of the geometry on `M2` from `cell`. The bounding box of the current cell may have been expanded, in which case some of the `M2` features may be off-screen (press the **Home** key to view the entire cell). Any objects previously existing on `M2` in the current cell are deleted before the operation, unless the **Don't clear** check box is checked.

10.10 The Properties Button: Property Editor Panel

The **Properties** button in the **Edit Menu** brings up the **Property Editor** containing commands for adding and modifying properties of objects. For the most part *Xic* does not use properties in physical layouts, but they provide important electrical information in schematic layouts, which is required when building a netlist or SPICE deck.

Clicking on a selected non-pcell instance with button 1 and the **Ctrl** key held will also bring up the **Property Editor**, if it is not already present.

When the **Property Editor** first appears, or upon pressing the **Activate** button in the panel, or if the **Properties** menu button is pressed with the **Property Editor** already visible but inactive, a command state begins where it is possible to list and edit the properties of selected objects. The command state is terminated by pressing the **Activate** button again, or pressing the **Properties** button in the **Edit Menu**, or pressing the **Esc** key, or by starting a different command. The **Property Editor** remains visible, but will go to an inactive state. The **Dismiss** button in the **Property Editor** will exit the command state if active, and retire the panel.

Unless stated otherwise, the descriptions of operations below apply only when the command state is active. When inactive, the presence of the **Property Editor** window has no effect, and other commands can be executed normally.

When the command mode becomes active, properties of one of the selected objects (if any) are shown in the text window of the panel. The objects are not generally shown as selected, but an internal list of objects that were selected before the command mode was started, or were clicked on with the command state active, is maintained. The object for which the properties are displayed is marked with a dotted outline around the object or a cross over the object. Clicking on the marked object will delete that object from the internal list, and another object's properties (if any in the list) will be shown. Clicking on an unmarked object will mark that object, add it to the list if it is not already there, and display its properties.

The **desel** button in the top button menu and other methods of deselection will clear the list of objects.

If the **Global** button in the panel is active, all objects in the list are shown as selected (blinking outline or symbol). The **Global** button allows manipulation of the properties of all objects in the list, not just the marked object.

When more than one device is in the list, the arrow keys can be used to cycle the marked object through the list.

When the **Info** button is active, clicking on an object will bring up or update the **Property Info** window, loaded with the properties of the object. This contains a listing identical to the **Property Editor**, however there are no buttons other than **Dismiss**. The object whose properties are listed in the **Property Info** window is marked on-screen similarly to the current object in the **Property Editor**, but with a different color.

When the **Property Editor** is active, clicking on an object with the **Shift** or **Control** key pressed will also bring up or update the **Property Info** window, whether or not the **Info** button is active.

The **Property Editor** and **Property Info** windows are drag/drop sources and receivers, meaning that one can drag properties from one window to another. This will apply the dragged property to the object associated with the drop window (the source object is not affected). Properties that must be unique, such as most electrical properties, will be replaced with the dropped property. Properties that are not unique will be added, without replacement. Only ordinary, user-modifiable properties can be copied in this manner. The prompt line, while in editing mode, is also a drop receiver for these windows.

The listing in either window shows the property number, a descriptive name in electrical mode, and the property string, for all properties attached to the current object. A property can be selected in the list by clicking on the text — it will be shown highlighted when selected. The current selection is used as input by many of the command buttons in the panel.

In the properties listing, color is used to distinguish the types of properties. The colors can be modified by setting the Special GUI Colors (see A.8.3) listed below. This can be done in the technology file, or with the **!setcolor** command.

variable	**default**	**purpose**
	black	internal properties
`GUIcolorHl1`	red	user-set `name` property
`GUIcolorHl2`	dark blue	physical mode pseudo-properties
`GUIcolorHl4`	sienna	ordinary (user-modifiable) properties

The value of the electrical mode `name` property is shown in a different color when the property is set. This property always exists, and it would not otherwise be obvious when viewing the listing when the `name` property has been set by the user, or is simply showing the name assigned by *Xic*.

The command buttons in the **Property Editor** allow addition, modification, and deletion of properties both globally (on all selected objects) or on the marked object. Those properties in the list marked as "internal" can not be modified. The physical mode pseudo-properties can not be edited, but can be added (with the **Add** button). In this case, no property is added, but the operation will cause some aspect of the object to change.

10.10.1 The Edit Button: Edit Property

The **Edit** button allows editing of the current property. If no property is selected in the text when the **Edit** button is pressed, the first user-modifiable property listed will become selected, and the text of that property will appear on the prompt line for editing. If a user-modifiable property was selected before the **Edit** button was pressed, the text of that property will appear on the prompt line. The up/down arrow keys will cycle through the editable properties listed in the window, selecting and placing the text on the prompt line in sequence. Also, clicking on an entry for a modifiable property in the window will select it and load its text into the prompt line.

The text in the prompt line can be edited, and pressing the **Enter** key completes the edit. The property listing will show the changes, if any. While editing, text from other windows can be inserted using drag/drop (from the property windows or the **File Selection** pop-up only) or with the window system cut/paste method.

When inserting text from property windows, hypertext references (see 3.1.2) are preserved. Hypertext entries can also be inserted in electrical mode by clicking on a device contact point or wire (node reference), on the '+' symbol of some devices (branch reference), or elsewhere on a device (name reference). Pressing **Shift** or **Control** while clicking on a device or subcircuit will bring up the **Property Info** window, whether or not editing is active.

For physical properties and value, param, and other electrical properties, the "long text" feature (see 7.9.5) is available. This is indicated by the presence of a small "**L**" button to the left of the prompt line, which appears when the prompt line cursor is in the first column. If this button is pressed, or **Ctrl-t** typed, a text editor window appears, loaded with the text of the property (if any). When a property is in long text format, the display listings will show only "`[text]`" as the content, and the prompt line will show the same string as a hypertext entry. In this case, just pressing **Enter** will bring up the text editor leaded with the "real" property text. This feature allows long, multi-line text blocks to be associated with properties.

The description thus far applies whether or not the **Global** button is set. With the **Global** button not set, when the **Enter** key is pressed to complete the editing, the property will be updated, and the text in the **Property Editor** will display the change. The operation, as with all operations described in this section, can be undone or redone with the **Undo**/**Redo** commands or **Tab**/**Shift Tab** keys.

If the **Global** button is set, the user will be prompted, in sequence, for a new string for each of the devices in the internal list. After the first prompt, the arrow keys and click-selection are disabled. Each device will be assigned a new property or a matching existing property will be replaced. For properties that can have more than one instance (other electrical properties and all physical properties) if the number and string of the original property shown highlighted in the **Property Editor** window match those of an existing property, that property will be replaced, otherwise a new property will be added.

10.10.2 The Add Button: Add New property

In physical mode, the **Add** button will produce a drop-down menu containing the following items.

nomerge
The **nomerge** choice will add a nomerge property (a property used by the extraction system) to the selected object or objects.

flatten
The flatten property applies to electrical and physical cells and instances. It is used during association and LVS to determine if the contents of the instance master should be logically promoted into

the containing cell (see 16.4). Although *Xic* can handle most hierarchy differences automatically and transparently, this property may be used when needed to force proper behavior.

If a flatten property has been applied to the master cell, then instances of the cell will be flattened, unless the instance also has the flatten property applied, in which case the instance will not be flattened. If the master does not contain a flatten property, then and instance will be flattened only if the instance has a flatten property applied. Thus, the flatten property of an instance reverses the effect of a flatten property applied to the master.

The FlattenPrefix variable, and equivalently the **Cell flattening name keys** entry area in the **Net and Cell Config** page of the **Extraction Setup** panel from the **Extract Menu**, provide another means of causing instances of cells to be flattened.

any

The **any** choice allows an arbitrary property to be added. This will initiate prompting for a property number and string to add.

In electrical mode, the **Add** button brings up a menu of property types that can be added. Selecting an entry will initiate prompting for the associated string. Any selection in the listing will be ignored. Unlike the case of the **Edit** button, the arrow keys and subsequent selection in the listing will not affect the prompt line.

With the **Global** button off, completion of editing by pressing **Enter** will "add" the new property to the current object. In electrical mode, properties other than the other property will be replaced if they exist, since there can be at most one such property. There can be arbitrarily many other properties, or properties of any number in physical mode. Such properties are always added and not replaced.

If the **Global** button is active, an identical copy of the property will be added to each of the devices in the internal list. This will be a replacement for electrical properties other than other, and an addition otherwise. Unlike the **Edit** button case, there is no individual prompting for a string for each device. The initial string (and number, in the case of physical mode) is added to each object.

In electrical mode, the **Add** menu contains buttons for the modifiable device and subcircuit instance properties listed below. Unless stated otherwise, there can be at most one each of the properties described below. This is enforced by *Xic*, i.e., attempts to add a second property of a given type will cause replacement, not addition.

name

The **name** button allows the modification of a name property. The name property specifies the device or subcircuit instance name to SPICE. Unlike the other user-settable properties, the name property always exists. If not explicitly set by the user, the device name will be generated internally. However, if a correspondence to an existing SPICE file is necessary, the name must be specified. *Xic* allows any name, however for the device to be recognized by SPICE, the name must start with the device's key letter as expected by SPICE. Deleting the name property simply reverts back to the internally generated name.

If an assigned name property conflicts with an internally generated name, the internally generated name will be updated so as to not conflict by appending "_N", where N is some integer.

model

value

The **model** and **value** buttons allow addition of a model or value property, respectively. Only one of the model or value properties should be used per device, as this really represents two different names for the same text field in SPICE output. One has the choice a supplying a device model or component value to the device, but not both. These properties generally apply to devices only, not subcircuits.

param

The param property is a catch-all for additional parameters found in the device and subcircuit instance lines in SPICE, such as initial conditions or device geometrical factors. The string will generally contain a list of *name*=*value* terms, each separated by white space. Only one param property is allowed.

devref

The devref property provides the name of the controlling device to current-controlled sources and the current-controlled switch. At most one devref property is allowed.

other

The **other** button allows addition of an other property. These properties have no significance to *Xic* and are not used in SPICE output. They can be used to store alternate values for the model, value, or param properties, or to store any other information desired by the user. There can be arbitrarily many other properties per device or subcircuit instance.

nophys

The **nophys** button allows addition of a nophys property. This property does not affect SPICE output, but specifies that the device or subcircuit instance has no physical implementation. When *Xic* is associating physical and electrical objects for extraction and LVS, a physical implementation will not be sought for objects with this property.

When the property is created, the user is prompted as to whether the device terminals should be shorted together during LVS. Devices that have the nophys property applied will be rendered using a different color than "normal" devices. See the description of LVS in 16.16 for a more complete discussion of the use of this property.

flatten

This will add a flatten property, which applies to electrical and physical cells and instances. See the description in the listing of physical properties above.

nosymb

The **nosymb** button is used to add a property to electrical subcircuit instances which forces them to be displayed as expanded, whether of not the master cell of the instance is symbolic. Instances with this property will behave in all respects as if the master were non-symbolic. Thus, instances of the same master can be displayed symbolically or not, in the same design. This property uses the same property number as the symbolic property applied to cells.

range

The **range** button is used to add a property to electrical device or subcell instances (other than terminal devices) that *vectorizes* the instance. The user is prompted for two non-negative numbers which define the subscripting range. Vectorized instances and connection rules are discussed in 4.2.9.

10.10.3 The Delete Button: Delete Property

If a modifiable property is selected in the list, pressing the **Delete** button will delete the property. If there is no selection, the user will be prompted. In physical mode, the user is requested to provide the number for the property or properties to delete. In electrical mode, the user is requested to provide a code consisting of any combination of the letters `n`, `m`, `v`, `p`, `o`, `y` to specify the properties to delete. If the response is "vp", for example, the value and param properties will be deleted. In the case of physical properties, all of the properties with the given number will be deleted (there can be more than one). Similarly, in electrical mode, if "o" is given, all other properties will be deleted.

If the **Global** button is not active, the properties are deleted from the current object. If the **Global** button is active, properties will be removed from all objects in the internal list. If a property was selected in the listing before the **Delete** button was pressed, and this is a physical property or other electrical property, only properties that match both the number and string (physical mode), or other properties that match the string (electrical mode) of the selected property will be deleted. If no selection is given, all properties that match the specification given will be deleted.

10.11 The Cell Properties Button: Edit Cell properties

The **Cell Properties** button in the **Edit Menu** brings up the **Cell Property Editor**, which is used to view and manipulate properties of the current cell. It is a simplified version of the **Property Editor** which is used to manipulate the properties of objects contained within the current cell.

The **Cell Property Editor** contains buttons to add, edit, and remove cell properties. In general, cell properties are assigned internally and can not be modified. The exceptions are the properties listed in the **Add** menu and further discussed below. Pressing the **Add** button brings up a pop-up menu containing entries corresponding to properties that can be set or modified by the user. Only the properties that are applicable to the current mode (physical or electrical) are active.

In physical mode, the entries listed below are available, allowing modification of physical cell properties (see D.1).

any

The **any** entry allows an arbitrary property to be assigned to the cell. The user will be prompted for a number and string for the property. These are arbitrary, however there are certain numbers that are reserved by *Xic* and will not be accepted. *Xic* will not use these properties, but they may be important for interfacing to third-party applications.

flags

The **flags** entry is used to set flags in the cell, notably the OPAQUE flag which causes the cell contents to be ignored during extraction.

flatten

The flatten property applies to electrical and physical cells and instances. It is used during association and LVS to determine if the contents of the instance master should be logically promoted into the containing cell (see 16.4). Although *Xic* can handle most hierarchy differences automatically and transparently, this property may be used when needed to force proper behavior.

If a flatten property has been applied to the master cell, then instances of the cell will be flattened, unless the instance also has the **flatten** property applied, in which case the instance will not be flattened. If the master does not contain a flatten property, then and instance will be flattened only if the instance has a flatten property applied. Thus, the flatten property of an instance reverses the effect of a flatten property applied to the master.

The FlattenPrefix variable, and equivalently the **Cell flattening name keys** entry area in the **Net and Cell Config** page of the **Extraction Setup** panel from the **Extract Menu**, provide another means of causing instances of cells to be flattened.

pc_params
The **pc_params** entry is used when defining parameterized cells (pcells, see 5.1). It is used to set or modify the parameter list associated with the pcell.

script
The **pc_script** entry is used when defining parameterized cells. It is used to set or modify the script which implements the pcell features.

In electrical mode, the following properties can be set from the **Add** menu.

param
Selecting the **param** button allows a param property to be added to the cell. The param property provides support for the subcircuit parameterization feature of *WRspice* (see the description of the `.subckt` line). The use of parameterization is briefly described in D.3.

other
Selecting **other** allows an other property to be added to the cell. These have no meaning to *Xic*, but might be of use to the user. Any number of other properties can be added.

virtual
Adding a virtual property will prevent the cell from being included in netlist output, most importantly SPICE output. The cell becomes a "placeholder", and the actual `.subckt` text, which is required to satisfy references, is included in the SPICE file by another means. For example, the cell might represent an opamp, and a `.include` line can be used to bring in the `.subckt` block representing the opamp, from a vendor model file.

flatten
See the description of the flatten property in the physical **Add** menu properties list above. The property has the same use in electrical mode.

For device cells, as would appear in the device library file, model, value, and param properties can be applied. When a device instance is placed, the instance will inherit copies of these properties. Instances of non-devices do **not** inherit a param property from the master. The model and value properties can not be applied with the **Cell Properties Editor**, but can be added to device masters with the **Library Device Parameters** panel (see 8.5).

This page intentionally left blank.

Chapter 11

The Modify Menu: Modify Geometry

The **Modify Menu** contains commands which alter the current design, supplemental to the side menu. Most of these commands have keyboard or mouse motion shortcuts, so an experienced user may not often use this menu.

The table below summarizes the commands that appear in the **Modify Menu**, including the internal command name and the command function.

Modify Menu			
Label	**Name**	**Pop-up**	**Function**
Undo	`undo`	none	Undo last operation
Redo	`redo`	none	Redo last undo
Delete	`delet`	none	Delete objects
Erase Under	`eundr`	none	Erase under objects
Move	`move`	none	Move objects
Copy	`copy`	none	Copy objects
Stretch	`strch`	none	Stretch objects
Change Layer	`chlyr`	none	Move object to new layer
Set Layer Chg Mode	`mclcg`	**Layer Change Mode**	Set layer change mode for move/copy

11.1 The Undo Button: Undo Operation

The **Undo** button in the **Modify Menu** reverses the operations performed in the current cell. These operations can be undone as long as the present cell is the current cell. Undone operations can be redone with the **Redo** command. Pressing the **Tab** key has the same effect as clicking on the **Undo** button, and **Shift-Tab** is equivalent to **Redo**.

By default, the last 25 operations can be undone. This can be changed with the variable UndoListLength, which can be set to a non-negative integer with the **!set** command. This sets the number of operations that are remembered. If set to zero, the list length is unlimited.

When *Xic* is waiting for text input to the prompt line, the **Undo** and **Redo** commands are disabled.

11.2 The Redo Button: Redo Last Undo

The **Redo** button in the **Modify Menu** will redo the last undone operation performed with **Undo**. This can also be accomplished by holding the **Shift** key and pressing the **Tab** key. Each undone operation is added to an internal list for possible redo. This list is cleared after any database-modifying operation which is not an undo.

When *Xic* is waiting for text input to the prompt line, the **Undo** and **Redo** commands are disabled.

11.3 The Delete Button: Delete Objects

The **Delete** button in the **Modify Menu** may be used to delete the selected objects. This is redundant, as selected objects can be deleted by pressing the **Delete** key.

11.4 The Erase Under Button: Erase Under Objects

The **Erase Under** button in the **Modify Menu** will erase the intersection area of non-selected objects with selected objects. The selected objects are not affected. This allows non-Manhattan holes to be cut in dark areas, for example. Suppose one needs a circular hole in a ground plane. Using this command, the task is simple. One would create the disk on some arbitrary layer where the hole is desired, select it, press **Erase Under**, then the **Delete** key to erase the disk object.

11.5 The Move Button: Move Objects

The **Move** button in the **Modify Menu** is used to move objects. This command is redundant, as objects can be moved with a basic button 1 operation. If objects are previously selected, the group will be moved. If no object has been selected, the user is requested to select an object to move. Responding to the prompts, the user points to a reference point, then to a destination point, using either hold and drag, or two clicks. If either the **Shift** or **Ctrl** key is held, the angle of translation is constrained to multiples of 45 degrees. The object is moved such that the reference point falls on the destination point. The orientation is altered according to the current transformation.

When the **Move** command is at the state where objects are selected, and the next button press would initiate the move operation, if either of the **Backspace** or **Delete** keys is pressed, the command will revert the state back to selecting objects. Then, other objects can be selected or selected objects deselected, and the command is ready to go again. This can be repeated, to build up the set of selections needed.

At any time, pressing the **Deselect** button to the left of the coordinate readout will revert the command state to the level where objects may be selected to move.

The current transform is saved in Register 0 then cleared after every move operation. Pressing the **Forward Slash** key will swap the current and saved transforms, allowing the last-used transform to be retrieved. The current transform is saved and cleared when **Deselect** is pressed, and cleared when the **Move** command exits.

The undo and redo operations (the **Tab** and **Shift-Tab** keypreses and **Undo**/**Redo** in the **Modify Menu**) will cycle the command state forward and backward when the command is active. Thus, the last command operation, such as initiating the move by clicking, can be undone and restarted, or redone if necessary. If all command operations are undone, additional undo operations will undo previous commands, as when the undo operation is performed outside of a command. The redo operation will reverse the effect, however when any new modifying operation is started, the redo list is cleared. Thus, for example, if one undoes a box creation, then starts a move operation, the "redo" capability of the box creation will be lost.

The substructure of cell instances being moved is highlighted to the depth shown in the main window. This facilitates alignment with other objects. One can change the display depth to reveal more or less of the substructure.

While in a move operation in physical mode, while the objects are ghost-drawn and attached to the pointer, pressing **Enter** causes the reference point to shift to the lower left corner of the bounding box containing the objects being moved. Pressing **Enter** will cycle the reference point through the corners of the bounding box, and back to the original reference location. Note that this allows objects that have somehow gotten off grid to be returned to the grid.

It is possible to change the layer of objects during a move operation. During the time that objects are ghost drawn and attached to the mouse pointer, if the current layer is changed, the objects that are attached can be placed on the new layer. Subcells are not affected.

How this is applied depends on the setting of the LayerChangeMode variable, or equivalently the settings of the **Layer Change Mode** pop-up from the **Set Layer Chg Mode** button in the **Modify Menu**. The three possible modes are to ignore the layer change, to map objects on the old current layer to the new current layer, or to place all objects on the new current layer. If the current layer is set back to the previous layer before clicking to locate the new objects, no layers will change. Note that layer change is only possible for "click-click" mode and not "press-drag".

Move operations can be also performed through the command line interface with the **!mo** command.

11.6 The Copy Button: Copy Objects

The **Copy** button in the **Modify Menu** is used to copy objects. In its simplest form, this command is redundant, as copies of an object can be made with basic button 1 operations. However, this command has an important and useful feature not available with the basic mouse operations: it is possible to copy objects from cells other than the one being edited.

Initially, the user is prompted for a replication count. This can be any positive integer. When the copy is performed, the replication specifies the number of copies made, with the translation incremented for each new copy. Thus, this facilitates creating many equally-spaced structures.

If objects are previously selected, the group will be copied to new locations. If no objects have been selected, the user is asked to select objects to copy.

Responding to the prompts, the user first clicks on a reference point, then to a destination, using a hold and drag, or two clicks. If either the **Shift** or **Ctrl** key is held, the angle of translation is constrained to multiples of 45 degrees. The copy is produced such that the reference point falls on the destination point. The orientation of the copied object is altered according to the current transformation. Multiple copies are made by simply clicking on additional destinations.

When the **Copy** command is at the state where objects are selected, and the next button press would initiate the copy operation, if either of the **Backspace** or **Delete** keys is pressed, the command will revert the state back to selecting objects. Then, other objects can be selected or selected objects deselected, and the command is ready to go again. This can be repeated, to build up the set of selections needed.

Ordinarily, if a sub-window is displaying a cell other than the current cell being edited, it is not possible to select objects in that sub-window. However, while the **Copy** command is active, if the sub-window has the same electrical/physical mode as the current cell, selections are allowed in the foreign sub-window.

Selections in a foreign window can be "picked up" just like objects selected in the main window. Outlines of the selected objects will be attached to the mouse pointer. They can be copied into the main window or a sub-window displaying the current editing cell by dragging or clicking twice.

Objects can be selected in various sub-windows and the main window simultaneously. Selections in sub-windows showing the current cell are in all respects equivalent to the main window. Use of the **Backspace** or **Delete** key method above is necessary to obtain selections in both the main window (and equivalent sub-windows), and sub-windows showing other cells. When the copy in initiated, only the objects from the cell in the clicked-in window (when objects are picked up) will participate in the copy operation.

Once objects have been picked up, whether copies have been placed or not, pressing either of the **Backspace** or **Delete** keys will revert the command state to the level before the objects were picked up. The user can then click in another window to pick up that window's selected objects, or in the same window to pick up the previous objects but with a different reference location.

At any time, pressing the **Deselect** button to the left of the coordinate readout will revert the command state to the level where objects may be selected to copy (in any mode-compatible window).

The current transform is saved in Register 0 after each copy is placed. Pressing the **Forward Slash** key will swap the current and saved transforms, allowing the last-used transform to be retrieved. The current transform is saved and cleared when **Deselect** is pressed, and cleared when the **Copy** command exits.

The undo and redo operations (the **Tab** and **Shift-Tab** keypreses and **Undo**/**Redo** in the **Modify Menu**) will cycle the command state forward and backward when the command is active. Thus, the last command operation, such as initiating the copy by clicking, can be undone and restarted, or redone if necessary. If all command operations are undone, additional undo operations will undo previous commands, as when the undo operation is performed outside of a command. The redo operation will reverse the effect, however when any new modifying operation is started, the redo list is cleared. Thus, for example, if one undoes a box creation, then starts a copy operation, the "redo" capability of the box creation will be lost.

The substructure of cell instances being copied is highlighted to the depth shown in the main window. This facilitates alignment with other objects. One can change the display depth to reveal more or less of the substructure.

The replication count feature is not available when copying objects from a foreign window, since the reference point is from another cell and is unlikely to be valid in the current cell. One copy of each selected object is created, at the click location or where dragging terminated, ignoring the replication count.

While in a copy operation in physical mode, while the objects are ghost-drawn and attached to the pointer, pressing **Enter** causes the reference point to shift to the lower left corner of the bounding box containing the objects being copied. Pressing **Enter** will cycle the reference point through the corners of the bounding box, and back to the original reference location.

It is possible to change the layer of objects during a copy operation. During the time that objects are ghost drawn and attached to the mouse pointer, if the current layer is changed, the objects that are attached can be placed on the new layer. Subcells are not affected.

How this is applied depends on the setting of the LayerChangeMode variable, or equivalently the settings of the **Layer Change Mode** pop-up from the **Set Layer Chg Mode** button in the **Modify Menu**. The three possible modes are to ignore the layer change, to map objects on the old current layer to the new current layer, or to place all objects on the new current layer. If the current layer is set back to the previous layer before clicking to locate the new objects, no layers will change. Note that layer change is only possible for "click-click" mode and not "press-drag".

When the **Copy** command terminates, any selected objects in foreign sub-windows are deselected.

Copy operations can be also performed through the command line interface with the **!co** command.

Example:
Suppose that you are editing cell B, and you would like to add a set of complicated polygons that you have already created in cell A.

1. Use the **Viewport** command in the **View Menu** to bring up a sub-window.
2. Use the **Load New** command in the sub-window **View** menu to display cell A.
3. Deselect any selected objects and instances.
4. Press the **Copy** button in the main window **Modify Menu**, and press **Enter** at the "Replication count" prompt.
5. Select the desired objects in the sub-window.
6. Click on a selected object in the sub-window, and drag or click again to copy the selected objects into the main window.

11.7 The Stretch Button: Stretch Objects

The **Stretch** button in the **Modify Menu** operates on polygons, wires, boxes, and labels. It enables moving of polygon and wire vertices, and box and label bounding box corners and sides. This command is somewhat redundant, as stretching operations can be initiated with basic button 1 manipulation, however the ability to select specific vertices to stretch is available only in the menu version of the command.

If no geometry has been selected, the user is asked to select objects to stretch. Otherwise, the stretch will be applied to currently selected objects.

After objects have been selected, specific vertices can be selected in boxes, polygons, and wires. The selection of vertices, which is available only in the menu version of the command, is accomplished by holding the **Shift** key, and clicking over a vertex, or dragging over one or more vertices. This operation can be repeated. Selecting a vertex a second time will deselect it. When a vertex is selected it is marked with a small highlighting box. When there are selected vertices, all selected vertices can be moved by clicking twice or dragging. The selected vertices will be translated according to the button-down location and the button up location, or the next button-down location if the pointer didn't move. While the translation is in progress, the new borders are ghost-drawn. While moving vertices, holding the **Shift** key will enable or disable constraining the translation angle to multiples of 45 degrees. If the **Constrain angles to 45 degree multiples** check box in the **Editing Setup** panel from the **Edit Menu** is checked, **Shift** will disable the constraint, otherwise the constraint will be enabled. The **Shift** key must be up when the button-down occurs which starts the translation operation, and can be pressed before the operation is completed to alter the constraint.

If a box is selected in the **Stretch** command, and one or more vertices of the box are selected by holding **Shift**, the vertices can be moved as for a polygon, and the box is converted to a polygon.

If no vertices are selected, the stretch operation applies to the nearest vertex of selected wires or polygons, or the nearest corner of a box. In this mode, boxes are stretched in a mode which preserves their rectangular shape. The user clicks on or drags to the new location, and the stretch is performed. If there are several objects selected, then the vertex closest to where the user points is taken as the reference vertex. This vertex is translated to the new location. In each of the other objects, the same transformation is applied to the vertex closest to the reference vertex. Thus, a group of wires, for example, can all be extended at once. During the operation, the **Shift** key and the **Constrain angles to 45 degree multiples** check box in the **Editing Setup** panel can be used to constrain the stretch angle as described above.

When the **Stretch** command is at the state where objects are selected, and the next button press would initiate the stretch operation or select a vertex, if either of the **Backspace** or **Delete** keys is pressed, the command will revert the state back to selecting objects. Then, other objects can be selected or selected objects deselected, and the command is ready to go again. This can be repeated, to build up the set of selections needed.

At any time, pressing the **Deselect** button to the left of the coordinate readout will revert the command state to the level where objects may be selected to stretch.

The undo and redo operations (the **Tab** and **Shift-Tab** keypreses and **Undo/Redo** in the **Modify Menu**) will cycle the command state forward and backward when the command is active. Thus, the last command operation, such as initiating the stretch by clicking, can be undone and restarted, or redone if necessary. If all command operations are undone, additional undo operations will undo previous commands, as when the undo operation is performed outside of a command. The redo operation will reverse the effect, however when any new modifying operation is started, the redo list is cleared. Thus, for example, if one undoes a box creation, then starts a stretch operation, the "redo" capability of the box creation will be lost.

The stretch operation works differently on Manhattan polygons than polygons containing nonorthogonal angles. For non-Manhattan polygons, a single vertex is moved, all others remain fixed. The stretch operation on Manhattan polygons is similar to the operation as applied to boxes, i.e., the corner and adjacent vertices are changed so as to keep the polygon Manhattan. A single vertex can be stretched arbitrarily either by selecting the vertex in the **Edit Menu Stretch** command, or by using the vertex editor in the **polyg** command.

If a wire end vertex is stretched to be coincident with the end vertex of another wire on the same layer with the same width, the wires will be merged, but only if the second wire is not selected.

11.8 The Change Layer Button: Change Layer

The **Change Layer** button in the **Modify Menu** allows the user to change the layer of the selected objects. All selected objects will be moved to the current layer. Objects must be selected before this command button is pressed.

11.9 The Set Layer Chg Mode Button: Set Change Mode for Move/Copy

This button brings up a panel which sets the layer change mode that applies to all move and copy operations, and to the **spin** command in the physical side menu. In these commands, when objects being moved or copied are ghost drawn as attached to the mouse pointer, it is possible to change the current layer. The operation is then completed by clicking at the new location in a drawing window.

The **Layer Change Mode** pop-up contains three "radio" buttons, which determine the response to the mid-command layer change.

Don't allow layer change
The layer change is simply ignored as it relates to the move/copy operation in progress. This is the default.

Allow layer change for objects on current layer
Any of the objects being moved/copied that are on the previous current layer will be moved or copied to the new current layer. Other objects are moved/copied normally.

Allow layer change for all objects
All objects will be moved or copied to the new layer.

The pop-up sets and tracks the state of the LayerChangeMode variable.

This is a tri-state variable. If not set, there will be no layer change in these commands. If set to the string "`all`" (case insensitive), then a layer change will apply to all objects being moved or copied. If set to anything else, including to nothing (i.e., as a boolean) then only objects on the previous current layer will be changed to the new layer.

This page intentionally left blank.

Chapter 12

The View Menu: Alter Presentation

The **View Menu** contains commands which alter the view shown in the drawing windows.

The table below lists the commands found in the **View Menu**. The internal command name is listed, as is the command function.

View Menu			
Label	**Name**	**Pop-up**	**Function**
View	`view`	none	Set view in window
Physical or Electrical	`phys` or `sced`	none	Switch mode
Expand	`expnd`	**Expand**	Show detail in window
Zoom	`zoom`	dialog	Change window scale
Viewport	`vport`	sub-window	New drawing window
Peek	`peek`	none	Show layers in area
Cross Section	`csect`	sub-window	Show layers in cross-section
Rulers	`ruler`	none	Add transient gradations
Info	`info`	**Info**	Show cell/object parameters
Allocation	`alloc`	**Memory Monitor**	Show memory statistics

12.1 The View Button: Select Cell View

The **View** button in the **View Menu**, and the **View** menu of sub-windows, produces a drop-down menu of view choices for the associated window. For each window, the last five views are saved in a list. In addition, up to five views can be saved by pressing **Ctrl-n**. These are assigned names consisting of the letters A–E. The drop-down menu entries are:

full center full view of cell
prev cycle view backwards
next cycle view forwards
A-E if view saved with **Ctrl-n**, set selected view

If the **View** command is "pressed" by a key sequence, the center full view is shown (same as for the **Home** key).

The view list is cleared whenever a new cell is displayed, or whenever the mode is changed for the window.

The **Ctrl-Shift-Right** arrow and **Ctrl-Shift-Up** arrow are accelerators for **prev**, **Ctrl-Shift-Left** arrow and **Ctrl-Shift-Down** arrow are accelerators for **next**, and **Ctrl-Shift-a** through **Ctrl-Shift-e** are accelerators for **A** through **E**.

12.2 The Physical Button: Show Physical Mode

The **Physical** button in the **View Menu**, and the **View** menu of sub-windows, available only when the window is displaying electrical mode, changes the display from electrical (schematic) mode to physical mode. In the main menu, this places *Xic* into an editing mode appropriate for physical representation. In the sub-windows, the **Physical** button changes the view only. Editing can not be performed in a sub-window whose mode is not that of the main window.

While in a **Push** (see 9.1), the cell currently being edited remains the current cell, but becomes top-level (i.e., not in a **Push**) in the new mode. If the original mode is returned to without editing a different cell, the **Push** stack is retained. If a new cell is edited in the new mode, through a **Push** or otherwise, the original **Push** context is lost. This context is also lost if the **Clear** function in the **Cells Listing** is invoked.

The present display mode can be made immutable, with certain side-effects, by setting the variable LockMode.

12.3 The Electrical Button: Show Electrical Mode

The **Electrical** button in the **View Menu**, and the **View** menu of sub-windows, available only when the window is displaying physical mode, changes the display from physical to electrical (schematic) mode. In the main menu, this places *Xic* into an editing mode appropriate for electrical representation. In the sub-windows, the **Electrical** button changes the view only. Editing can not be performed in a sub-window whose mode is not that of the main window.

While in a **Push**, the cell currently being edited remains the current cell, but becomes top-level (i.e., not in a **Push**) in the new mode. If the original mode is returned to without editing a different cell, the **Push** stack is retained. If a new cell is edited in the new mode, through a **Push** or otherwise, the original **Push** context is lost. This context is also lost if the **Clear** function in the **Cells Listing** is invoked.

The present display mode can be made immutable, with certain side-effects, by setting the variable LockMode.

12.4 The Expand Button: Expand Subcells

The **Expand** button in the **View Menu**, and the **View** menu of sub-windows, brings up the **Expand** pop-up, which controls the expansion of subcells in the window. All geometry is shown in an expanded cell, whereas only the bounding box and possibly a name label are shown in the unexpanded state. If the cell happens to be an array, the bounding box in the unexpanded state is shown as a dashed line. Ordinary instances have a solid line bounding box.

The label shown in unexpanded physical instances is by default the instance name, which consists of the master name followed by a colon separator and an index number. The index is a 0–based sequence for instances with a particular master. The index count advances by the size of the array for arrayed instances, leaving room in the sequence for individual elements. The index is in database order (top to bottom then left to right of the upper left corner of the instance bounding box), and is stable and reproducible as long as instance sizes and placement locations remain the same. If the boolean NoInstnameLabels variable is set, the label will display the master name only, which was the behavior in *Xic* releases prior to 4.3.3.

After pressing **Expand**, the pop-up appears. The pop-up contains a text entry area, a number of buttons which push specific text into the entry area, an **Apply** button, a **Dismiss** button, and a **Help** button. When the pop-up first appears, it is given the keyboard focus. Under most (if not all) window managers, one should be able to type into the text entry area immediately. Pressing the **Enter** key is equivalent to pressing the **Apply** button. Thus, one can quickly change the expansion status entirely with the keyboard accelerators (the change will apply to the window containing the pointer).

For example, the default keypress mapping applies **Ctrl-x** to the **Expand** button, so typing

Ctrl-x 0 Enter

will set the expansion level to 0, and

Ctrl-x a Enter

will set the expansion level to "all".

The functions of the symbols which are recognized in the text string will be described below. The buttons which push text into the entry area avoid the need for typing. These are:

+	set to '+' (there can be multiple +'s added)
−	set to '−' (there can be multiple −'s added)
All	set to '`all`'
0-5	set to '`0`' – '`5`'
Peek Mode	set to '`p`' (available from main window only)

Pressing the **Apply** button will pass the expansion string to the internal expansion control function.

The characters which are recognized in the string are the letters `a`, `n` for "all" and "none", one or more + or − symbols (not mixed) which will increment or decrement the hierarchy depth of expansion, a + or − followed by an integer, which will increase or decrease the level by that integer, or simply an integer, which will set the hierarchy depth to that integer. Setting the hierarchy depth to zero is the same as "none". All subcells up to the hierarchy depth are shown expanded.

Each drawing window has its own expansion parameters and **Expand** button. When a sub-window is created, it inherits the expansion status of the main window. The expansion depth entered applies only to that window.

12.4.1 Peek Mode

If the **Expand** button from the **View Menu** is selected, there is an additional feature available: peek mode, which is entered by returning `p`. This should not be confused with the **Peek** command in the **View Menu**. In peek mode, the expanded status of individual cells and subcells can be set by clicking or dragging with button 1. Only cells below the current expansion depth are affected, i.e., those that are normally displayed as unexpanded. Thus, peek mode has no effect if the expansion depth is set to "all". Clicking on an unexpanded cell, or dragging such that the cell is enclosed within the drag rectangle, will cause that cell to be shown as expanded (to one level) in the window where the button down event occurred. The process can be repeated to expose cells arbitrarily deep in the hierarchy. If the **Shift** key is held during the pointing operation, previously expanded cells are unexpanded. This applies only to cells below the expansion depth. Note that unlike a standard selection operation, in peek mode one can address subcells below the first hierarchy level, so long as the parent cell is shown expanded.

Peek mode works by setting a flag in the instance descriptor of a subcell. Instance descriptors are stored in the parent cell. Suppose that a design contains multiple instances of cell B, each of which contains a left and right instance of cell A. In peek mode, for example, if the left instance of A is made to be expanded in an instance of B (which of course is also expanded), this expansion of A will appear in all instances of B which are expanded, not just the one clicked on. This is a consequence of the hierarchical nature of the database, where each instance of B represents the contents of B, which includes the instance of A with the flag set.

In peek mode, the operation applies to any window in which the pointer was located when button one was pressed. The result will be consistent with the expansion depth of the particular window. While in peek mode, certain keyboard commands can be applied, which will affect the window where the pointer was located when the key press occurred. The + and − keys increment and decrement the expansion depth, a number key will set the expansion depth, **a** will set the depth to "all", and **n** will set the depth to "none", as will **0** (zero). Each window has independent expansion parameters. Setting the expansion depth to zero by pressing **0** or **n** will clear the peek mode display flags. Otherwise, the expansion depth and the peek mode display of cells are independent.

In electrical mode, symbolic cells can be shown as expanded, with a miniature rendition of the actual circuit inside the symbolic bounding box area, in peek mode. Click on the symbolic cell to expand, **Shift**-click to unexpand. Wires connecting the circuit connections to the symbol terminals are added. This rendition is for visual purposes only. If a subcell placed in symbolic mode is later changed to non-symbolic mode, the view of the parent cell is likely to look horrid, since the subcircuits will probably overlap. The peek mode feature allows viewing of the underlying circuit without this problem.

The expansion status of a given subcell in a window is retained after exiting peek mode, and after canceling a sub-window. Setting the expansion to "none" clears all expansion in peek mode.

12.5 The Zoom Button: Zoom In/Out

Pressing the **Zoom** button in the **View Menu** or the **View** menu of sub-windows brings up the **Set Display Window** pop-up. This pop-up can change the scale (zoom) the window, or set a new display region.

To change the scale, enter a factor into the **Zoom Factor** entry area, and press the associated **Apply** button. Factors greater that 1.0 will zoom out.

Alternatively, one can enter the center x and y values and width (all in microns) of a new region to display. The coordinates are relative to the origin of the displayed cell. The width is the displayed width of the region to be displayed in the window, the displayed height will depend on the drawing window's aspect ratio. Pressing the associated **Apply** button will redisplay the new location.

The windowing parameter entries are pre-loaded with the current window parameters, and track any changes made when the pop-up is visible.

The right mouse button (button 3) can also be used to zoom, as can the numeric keypad + and − keys which zoom in and out by a factor of two, or by ten percent if **Shift** is also held.

In the case where the panel is brought by from a sub-window that is displaying a cross-section, there are additional controls that allow adjustment of the vertical scaling used in the cross section display.

This control group has its own **Apply** button. The **Auto Y-Scale** check box disables the automatic vertical scaling when set. The automatic scaling maintains full view of the entire layer stack independent of the zoom factor. The **Y-Scale** entry allows a vertical scale factor to be entered, for use when displaying the cross section. When the **Apply** button is pressed, the cross-section is redrawn with the settings given. The XSectNoAutoY variable will be set or unset to reflect the state of the check box. The XSectYScale variable will be set to the scale factor if it is other than unity. See 12.9 more more information.

12.6 The Viewport Button: Create Sub-Window

The **Viewport** button in the **View Menu** brings up a sub-window, which is a display window similar to the main drawing window. The user is requested to point at the diagonal endpoints of the region to be displayed in the sub-window. Each viewport contains a menu of attribute buttons which apply to that window only. In particular, the sub-window can display cells in either electrical or physical mode, however editing operations are only possible if the sub-window mode and cell match those of the main window.

The sub-window has a set of menus which control attributes which can be set on a per-window basis. When the cursor is in a sub-window, characters entered are delivered to that sub-window, and an unambiguous sequence match will select a sub-window button. Matches are looked for in the sub-window menu, the main menu, and any pop-up menus, in that order.

The **View** menu button commands are mostly analogous to the commands found in the main **View Menu**, however there are a few entries in the **View** menu that have no analogs in the main menu.

Sub-Window View Menu			(additional)
Label	**Name**	**Pop-up**	**Function**
Dump To File	`wdump`	text entry	Dump window to image file
Show Location	`lshow`	none	Show position in main window
Swap With Main	`swap`	none	Swap contents with main window
Load New	`load`	none	Load cell or file for viewing

The **Attributes** menu is identical to the **Main Window** sub-menu found in the main **Attributes Menu**. The functions are the same (see 13.11 but apply to the sub-window only. When a new sub-window appears, it inherits the current attribute settings of the main window.

Sub-Window Attributes Menu			
Label	**Name**	**Pop-up**	**Function**
Freeze Display	`freez`	none	Suppress redisplay
Show Context in Push	`cntxt`	none	Show context in subedit
Show Phys Properties	`props`	none	Show physical properties
Show Labels	`labls`	none	Show labels
Label True Orient	`larot`	none	Show labels transformed
Show Cell Names	`cnams`	none	Show cell names
Cell Name true Orient	`cnrot`	none	Show cell names transformed
Don't Show Unexpanded	`nouxp`	none	Don't show unexpanded subcells
Objects Shown	`objs`	none	Object display control
Subthreshold Boxes	`tinyb`	none	Show outline of subthreshold cells
Set Grid	`grid`	**Grid Parameters**	Set grid parameters

If a cell shown in a sub-window is the cell shown in the main window, with the same mode (physical or electrical), then all editing operations will work in the sub-window as well as the main window. The sub-window will display all highlighting, terminals, and other special markings. If the sub-window shows a different cell, then in that window selections and editing are not possible, and no highlighting or special markings are shown.

When the **Viewport** command is used to create a new sub-window, the sub-window will initially show the same cell as the main window.

The sub-windows are sensitive as drop-receivers from the file manager and other listing pop-ups. File or cell names can be dragged from the listing pop-up and dropped in a sub-window, which will cause that cell/file to be opened and displayed in the window.

The **Dump To File** button in the **view** menu of sub-windows will dump the contents of the window to a disk file. When pressed, a file name will be solicited, and the contents of the sub-window will be dumped to the file. The filename extension determines the file type: jpg, tiff, png, etc. This provides a mechanism for obtaining printable output from the **Cross Section** views. The dumped bitmap will be the same size as the window.

This feature makes use of the `imsave` system, which is also used in the Image print driver (see 8.6.2).

When the **Show Location** button in the sub-window **View** menu is active, and both the sub-window and the main window are in physical mode and displaying the same cell, an outline box is drawn in the main window around the area displayed in the sub-window. This indicates the position of the sub-window display, assuming that the sub-window is showing a zoomed-in part of the display in the main window.

The **Swap With Main** button in the sub-window **View** menu will swap the cells, display modes, and views between the sub-window and the main window. This has the effect of making the cell displayed in the sub-window the current cell, allowing it to be modified.

The **Load New** button in the sub-window **View** menu will prompt for a new cell or file to display in the window. The command will prompt the user for a file/cell name, in the manner of the **Open** command. The given file/cell will be opened for display in the sub-window.

12.7 The Peek Button: Show Layer Composition

The **Peek** button in the **View Menu** asks the user to define a rectangular area with pointer clicks ar drag, and then redisplays the area slowly so that underlying layers can be seen. It also prints the names of layers found (only physical objects are considered, not labels). The delay, which defaults to .4 second per existing layer, can be reset with the PeekSleepMsec variable, which can be set to the delay time in milliseconds.

12.8 Three-Dimensional Layer Sequence Generator

Xic contains functionality to generate sequenced three-dimensional layer stack representations of layout geometry. This capability is employed by the **Cross Section** command in the **View Menu** which displays the layout in cross-section, and in the interface to external capacitance and inductance extraction programs.

In order to use this functionality, the appropriate keywords must have been applied to the layers.

Most of the setup parallels that required for the extraction system.

Layers that participate must satisfy the following constraints.

- The layer must **not** have the Symbolic technology file keyword given.
- The layer must be visible in the layer table.
- The layer must have a nonzero thickness applied with the Thickness technology file keyword.
- The layer must be a conductor or an insulator. A conductor layer has one of the following:
  1. The Conductor keyword is given, explicitly or implicitly if one of the Routing, GroundPlane, GroundPlaneClear, Contact, or their aliases is given.
  2. Any of the Rsh, Rho, Sigma, or Lambda keywords is given with a positive value.

  An insulating layer has either the Via or Dielectric keyword given, and also the EpsRel keyword given with a value of 1.0 or larger.

The technology file keywords are normally applied in layer blocks of the technology file, but can be applied from within *Xic* as well. The **Tech Parameter Editor** from the **Edit Tech Params** button in the **Attributes Menu** allows editing of the associated keywords.

The polarity of the layer will be obtained from the keywords applied to the layer. The layer will be considered dark field if any of the DarkField, Via, or GroundPlaneClear keywords are given to the layer. In this database, each layer represents the true presence of physical material which is the inverse of the normal presentation if the layer is dark field. For example, a via showing as a small colored square in a normal layout window is actually a hole in an otherwise continuous insulating film. The layer represented in this database will contain shapes representing the areas where the physical film is present. This can be seen in the **Cross Section** displays, where the dark field layers will be displayed with the inverse polarity from normal drawing windows.

All layers can be set to planarize, or not. This is an important difference from the original interface, which always assumed planarization. If a layer is not planarizing, it conforms to the underlying topology, which is translated exactly to the top surface of the layer. If the layer is planarizing, representing the layer three-dimensionally is accomplished as follows. Initially, the shapes on each layer are decomposed into a non-overlapping collection of trapezoids.

1. Consider the non-planarized representation of the layer, which is composed of trapezoids in the X-Y plane (parallel to the substrate), each with an elevation (distance from the substrate in the Z direction) and constant thickness. Each trapezoid represents an area of the layer material at constant elevation. Sort through the trapezoids, and find the bottom surface value with the highest elevation.
2. From the maximum lower surface elevation, set the "plane" value for the layer. Set the top elevation of all trapezoids to the plane value plus the film thickness value. All trapezoids will have the same top elevation, but in general will have differing bottom elevation. The actual film thickness varies, but is equal to or greater than the thickness value of the layer.

Thus, for a planarizing layer, the top surface of the layer will be a plane at the highest point on the top surface of the layer below plus the layer thickness. The bottom surface of the layer will conform to the top surface of the layer below by filling in where material of the layer below is not present.

By default, all Conductor (and the rest of the keywords that imply Conductor) and Via layers are planarizing. This is normal for a modern metal stack in a semiconductor process. The Dielectric layers, and conductor layers that don't have Conductor applied implicitly or explicitly, but therefor must have Rho or Rsh applied, are not planarizing by default.

However, if the NoPlanarize variable is set, or the global technology file attribute of the same name is given, by default no layers will be planarizing.

There is a technology file layer block keyword which provides planarization control for each layer. It overrides defaults and state of the NoPlanarize variable.

`Planarize` [y—n]
: This specifies whether or not a layer is "planarizing". The Planarize keyword can be applied to prevent planarization of layers that are planarized by default, or to force planarizing of layers that don't normally have this property.

The database is not intended for large collections of objects, and processing time is near quadratic in the number of database trapezoids. By default, the total trapezoid count is limited to 10000. Attempts to exceed the limit will fail, causing the command using the database to also fail, with an appropriate error message.

The limit can be modified with the Db3ZoidLimit variable. This variable can be set to an integer 1000 or larger to reset the limit. If not set, the limit of 10000 applies.

12.8.1 Layer Sequencing

By default, all conducting and insulating layers are assumed to stack in layer table order, starting at the substrate. We therefor ignore the upper and lower conductor references in Via layers. Any layer order is acceptable, there is no constraint regarding adjacency of layer types, but of course the layer table order must match the physical order.

However, we allow for the case that Via layers are out of sequence in the layer table. They will be moved to the correct position in the stacking order used by the interface. Since layers are rendered bottom-up in drawing windows, having a Via layer positioned above the referenced top conductor in the layer table would cause the via to be drawn on top of the metal, probably enhancing visibility. The correct sequence, of course, would place the Via layer below the top referenced conductor.

The LayerReorderMode variable can be set to allow Via layer repositioning. The variable is set to an integer in the range 0–2.

0
: No repositioning is done, the same as if the variable is not set.

1
: Consider the via references. A layer can have multiple Via keywords, each specifying a pair of conducting layers which are to be connected through a hole in this layer. For each pair, we identify the bottom conductor by its position in the layer table relative to the other referenced conductor. For each of the bottom consuctors, we find the one that is highest in layer table order, and move the Via layer to just above this layer.

2
: Similarly, we identify the lowest of the upper conductors, and move the Via layer to just below this layer.

Once the layers are recognized and sequenced, a three-dimensional representation of the layers found within a specified area is constructed. This representation is then available for such useful things as displaying a cross section, or building up an input file for a parameter extraction program.

12.9 The Cross Section Button: Show Cross Section

The **Cross Section** button in the **View Menu** brings up a special sub-window which displays a cross sectional (side) view of the layers under an arbitrary line. After pressing the command button, the user is asked to define a line, which can be done by clicking twice or dragging. If the line covers any geometry (which may be implied by dark field layers), a sub-window showing the cross sectional view will appear. The process can be repeated. Pressing the **Esc** key will exit the command.

All geometry under the line will be shown, without regard to cell hierarchy.

If the **Constrain angles to 45 degree multiples** check box in the **Editing Setup** panel from the **Edit Menu** is checked, the angle is constrained to multiples of 45 degrees. If not checked, the angle is unconstrained, but snaps to multiples of 45 degrees when the angle is close. In either case, pressing the **Ctrl** key removes the constraint.

The endpoints initially do not snap to grid points. The period ('.') key toggles snapping to grid of the endpoints, when defined with the mouse or other pointing device.

The endpoints are saved in persistent storage, and the previous cross section can be repeated by pressing **Enter** while the command is active, even if the command terminated after the last cross section was displayed. One can experiment with different thicknesses or planarizing behavior of the layers, and easily compare cross sections from the same line, using this feature. If the current cell has changed, the stored endpoints will have no effect.

The display makes use of the three-dimensional layer sequencing database described in 12.8 to build up a three-dimensional representation of the geometry along the line. This requires that the layers are appropriately set up in the technology file. There are few defaults, and this command will not work without proper setup.

The layer thickness shown can be set with the CrossThick technology file keyword. This can be applied in the physical layer blocks of the technology file, or can be set or edited from the **Tech Parameter Editor**.

If CrossThick is not set, the displayed thickness will be taken from the Thickness parameter, which must be set to a nonzero value in any case. This is the physical film thickness. In cases where this is too thick or thin for convient viewing in cross-section, the CrossThick keyword can be set to provide an overriding thickness used in the display.

The layer shown in the cross section is always true polarity, showing where the material will exist on the substrate. This is the inverse of the normal drawing windows when the layer is dark field. A via, for example, which appears as a colored square in the main window, should appear as a hole in cross section, since the painted area actually represents lack of insulating material. This is a "dark field" layer.

By default, the display uses **Auto Y-Scale** mode, where the total displayed thickness of the layer stack is a little less than the display window height. This scaling is used for any magnification, which of course has the usual effect in the horizontal direction.

The Y-scale can be manipulated by the panel brought up from the **Zoom** button in the **View** menu of the cross-section display window. In this configuration, the **Set Display Window** panel has a set of controls for adjusting the Y-scale. The **Auto Y-Scale** check box sets whether of not the automatic scaling is used. This corresponds to the XSectNoAutoY variable.

The **Y-Scale** entry area allows a scale factor to be entered. In **Auto Y-Scale** mode, this will change the displayed layer stack height relative to the window height. Otherwise, this will be a constant linear scale factor applied in the Y direction.

Note that the grid lines will be shown taking into account the current Y scale, and the cells will therefor not, in general, be square. The scaling is also accounted for in the gradations computed for rulers that may be applied to the cross-section window.

12.10 The Rulers Button: Create Rulers

The **Rulers** button in the **View Menu** provides a facility for creating rulers. Rulers, available in physical mode, are visible calibrated gradations which indicate physical distance in microns. Rulers are often convenient for measuring distances, and in hard copies to indicate size scale.

When the **Rulers** button is on, rulers can be created by clicking twice at the endpoints of the ruler, or by pressing and dragging, where the ruler will extend from the press and release points. The ruler will only be visible in the window where the first button press occurred, and only rulers in the main window will be visible in hard copies. Rulers can be created in any physical-mode window, including cross-section and cell hierarchy digest displays.

The computed distance between endpoints of the ruler is printed in the prompt area after a ruler is created. This can be used to accurately measure the distance between two points.

Rulers remain visible until another cell is edited, or until deleted. The rulers in effect for a certain cell are remembered, so that upon returning to a previously edited cell, the rulers previously in effect will be visible.

Presently, rulers exist only in memory, and are not saved to disk with cell data.

Rulers can be deleted, while the **Rulers** button is on, by pressing the **Delete** or **Tab** (undo) keys, and are deleted in reverse order of creation, but only in the window that has keyboard focus.. Rulers associated with the current cell can be deleted at any time with the **!dr** command.

When a ruler is being created, the ghost-drawn vector which appears when creating a ruler indicates the side which will have the gradations. The side with the gradations can be toggled by pressing the '/' or '\' keys.

If **Shift** is pressed during completion of a ruler, the endpoint will be the start point of a new ruler, and the calibration in the new ruler will be an extension of that in the current ruler. Thus rulers can be "chained" around an object to measure the periphery.

If the **Constrain angles to 45 degree multiples** check box in the **Editing Setup** panel from the **Edit Menu** is checked, the angle is constrained to multiples of 45 degrees. If not checked, the angle is unconstrained, but snaps to multiples of 45 degrees when the angle is close. In either case, pressing the **Ctrl** key removes the constraint.

By default, the tiny snap-box near the mouse pointer, where ruler endpoints can be placed, is constrained to the current snap grid, as is true elsewhere in *Xic*. The period ('.') key toggles snapping to grid points while in the **Rulers** command. When not snapping, the snap box follows the mouse pointer, allowing per-pixel resolution.

Additionally, if the window shows a normal layout (not CHD or cross section) the endpoints will by default snap to nearby edges and vertices. This is the same edge snapping as is controlled from the

Edge Snapping group in the **Snapping** page of the **Grid Setup** panel (pressing **Ctrl-g** will produce this panel). However, the **Rulers** command has its own defaults, which are active while the **Rulers** command is active. The default is to allow off-grid locations, and non-Manhattan edges, and to snap to both edges and the path (central spine) of wires. Any of these settings can be changed, and edge snapping disabled, from the **Grid Setup** panel. Changes will apply only in the **Rulers** command. Edge snapping applies whether or not grid snapping is enabled.

12.11 The Info Button: Display Information About Objects

The **Info** button in the **View Menu** brings up a an **Info** window, which can display information about the current cell or any visible object. This command can also facilitate pushing the editing context to specific locations within the hierarchy.

Alternatively, if any objects are selected when the command is given, information about all selected objects can be dumped to a file. When the command is entered, if there are selected objects, the user is prompted whether to dump the info to a file. If '`y`' is given to the prompt, the user is asked for a file name, and is given the option of viewing the file when the dump is complete.

If there is no prompt, or '`n`' is given at the prompt, any selected objects become deselected, and "info mode" becomes active. Objects and unexpanded subcells can be clicked on, and information about an object will be displayed in the text window that will appear. The chosen object will be highlighted in the display.

Although the clicking/highlighting operation is superficially similar to normal selection, in fact there are important differences. However, the layer selectability and object type selectability flags for normal selections are observed.

1. Any object or unexpanded subcell visible in the drawing window can be chosen, at any depth in the cell hierarchy. In normal selection, only objects and subcells of the current cell can be selected. The "selection" mechanism tracks the expansion depth of the display window, including peek mode if active (see 12.4). Any object that is visible, or any subcell that is shown as unexpanded, can be chosen by clicking on the object in the window.
2. Only one object or unexpanded subcell can be highlighted at a time. This is the item whose information is shown in the window. To highlight a different object at the same location, click multiple times. A different object will be highlighted on each click.

The information shown in the window will include the name of the cell that contains the chosen object, and a "back trace" of containing subcells in the hierarchy up to the current cell. Thus, one can easily determine the cell which "owns" an object in the display.

While an object is selected, pressing **Enter** will toggle selection of all or other objects in the cell containing the original selected object. The objects selected will respect the settings of layer-specific selection mode, and object type selectability, as set in the **Selection Control Panel**. In this state, the text in the **Info** window will describe the instance containing the selected objects. Pressing **Enter** again will revert to the previous state. This is useful for determining which objects belong to a particular cell instance.

When an object is highlighted, initiating the **Push** command in the **Cell Menu** will push the editing context to the cell containing the chosen object. That is, the current cell becomes the cell containing the object, in the context of the instance in the chosen location. The objects in this cell can then be edited. The **Pop** command in the **Cell Menu** can be used to pop the editing context back up the hierarchy to the original current cell. This can be a useful way to navigate through a complex hierarchy, while editing a layout.

If the **Shift** key is held while the user clicks anywhere in a drawing window showing the current cell, or if the click occurs outside of the current cell bounding box, information about the current cell is shown, rather than information about a chosen object.

If the user clicks in a sub-window that is displaying a cell that is not the current cell, information about the cell will be shown. It is not possible to select objects in this case.

The information shown in the text window contains items such as the object type and bounding box, as well as details specific to the type of object. For objects not in the current cell, coordinates are usually shown relative to the object's containing cell, as well as those reflected to the current cell.

By default, dimensions are given in microns. If the variable InfoInternal is set (with the **!set** command) then dimensions are given in internal units (usually 1000 units per micron).

The text window contains two buttons. The **Dismiss** button removes the pop-up and exits info mode. The **Activate** button, which is initially active, can be used to exit and reenter info mode, while the pop-up remains visible.

When applied to polygons, the **Info** command performs reentrancy tests, and a message is added if the condition is found, i.e., if the polygon can not be rendered unambiguously.

Similarly, when applied to wires, the wire is checked for certain properties that might cause trouble. See the description of the **!wirecheck** command in 19.14.13 for a description of the flag keywords that might appear in the info text.

The **Info** pop-up is also made visible by the **Info** button in the **Cells Listing** panel brought up by the **Cells List** button in the **Cell Menu**. When this button is used, previously selected objects are ignored, and there is no provision for dumping to a file. The **Info** window will provide information on cells selected in the cells listing, or on objects selected in the drawing windows.

12.12 The Allocation Button: Show Memory Allocation

Pressing the **Allocation** button in the **View Menu** brings up the **Memory Monitor** pop-up. This displays the number of cells in memory, the total dynamic memory in use by the program, and system limits on dynamic memory. While visible, the pop-up is refreshed every few seconds.

The maximum memory that can be used by the program before a fault occurs is not well defined, and may be much less that the limits, depending on what other programs are running, the actual size of the swap space, and other factors. The limits are either system defaults, or values set with the `limits`(1) shell command (Unix/Linux). The "hard" and "soft" values are those returned by the system call, and have different interpretations under different Unix versions.

Chapter 13

The Attributes Menu: Set Display Attributes

The **Attributes Menu** contains commands primarily for modifying the presentation format of *Xic*. Each sub-window has an independent set of attributes, which are initially set to those of the main window.

The table below summarizes the command buttons found in the **Attributes Menu**, listing the internal name and command function.

Attributes Menu			
Label	**Name**	**Pop-up**	**Function**
Save Tech	`updat`	none	Save technology file
Key Map	`keymp`	none	Create keyboard mapping file
Define Macro	`macro`	none	Define a keyboard macro
Main Window		Attributes sub-menu	Set main window attributes
Set Attributes	`attr`	**Window Attributes**	Set misc. attributes for drawing windows
Connection Dots	`dots`	**Connection Points**	Show connection dots in schematics
Set Font	`font`	**Font Selection**	Set text fonts used
Set Color	`color`	**Color Selection**	Set layer and other colors
Set Fill	`fill`	**Fill Pattern Editor**	Set layer fill patterns
Edit Layers	`edlyr`	**Layer Editor**	Add or remove layers
Edit Tech Params	`lpedt`	**Tech Parameter Editor**	Edit technology parameters

The **Connection Dots** button appears only when the full *Xic* feature set is enabled, not in the *XicII* or *Xiv* virtual products.

The sub-menu brought up by the **Main Window** button is identical to the **Attributes** menu in the sub-windows produced from the **Viewport** button in the **View Menu**. The settings apply to the main window.

Sub-Window Attributes Menu			
Label	**Name**	**Pop-up**	**Function**
Freeze Display	`freez`	none	Suppress redisplay
Show Context in Push	`cntxt`	none	Show context in subedit
Show Phys Properties	`props`	none	Show physical properties
Show Labels	`labls`	none	Show labels
Label True Orient	`larot`	none	Show labels transformed
Show Cell Names	`cnams`	none	Show cell names
Cell Name true Orient	`cnrot`	none	Show cell names transformed
Don't Show Unexpanded	`nouxp`	none	Don't show unexpanded subcells
Objects Shown	`objs`	none	Object display control
Subthreshold Boxes	`tinyb`	none	Show outline of subthreshold cells
No Top Symbolic	`nosym`	none	Electrical only, don't show top cell as symbolic
Set Grid	`grid`	**Grid Setup**	Set grid parameters

13.1 The Save Tech Button: Update Technology File

The **Save Tech** button in the **Attributes menu** will pop up a small panel, from which one may write an updated technology file to disk. This file provides setup information to *Xic*, and is read on program start-up. The file will reflect the current settings of the configurable attributes, layers, etc.

The **Write Tech File** panel has three radio buttons that select how parameters that are currently set to program defaults will be handled. By default, keywords which would specify a default value are omitted from the file, providing a more compact file. However, it may be useful to include the defaults in the file, to facilitate future hand-editing. If **Comment default definitions** is selected, these lines will be added to the technology file as comments. If **Include default definitions** is selected, these lines will be added as active text.

An entry area allows the user to change the name of the file to write. The default is to use "`xic_tech`" with an extension, if any, that was given to *Xic* on startup with the `-T` option, written in the current directory. The base name must be "`xic_tech`" and the file must be located in the library search path in order to load the new file when *Xic* is restarted.

The radio buttons track and set the status of the TechPrintDefaults variable.

13.2 The Key Map Button: Create Key Mapping File

Several of the keys which *Xic* uses are not found on all keyboards, or they may return a different code than *Xic* expects. *Xic* contains a built-in facility for remapping keys from the keyboard to the functions expected in *Xic*, through the **Key Map** button in the **Attributes Menu**. The user is prompted to press keys on the keyboard that will correspond to the various special keys such as **Page Up**, **Page Down**, **Home**, and numeric keypad **Add** and **Subtract**. Only special function keys can be remapped, not the standard character entry keys, or modifier keys (e.g., **Shift**, **Control**) and not the function keys 1–12. If there is no response when a key is pressed and the pointer is in a drawing window, then that key can not be mapped. To skip mapping of a key, press **Enter**. To abort, press **Esc**. When the prompting is finished, a file will be created in the current directory named `xic_keymap` by default. The user is given the option to alter this name.

The file can be moved to a safe place if desired. The keyboard mapping is not actually performed until this file is read by *Xic*, which must be done explicitly. This differs from *Xic* releases earlier than 4.0.0, which attempted to automatically read a created mapping file. There are two ways to accomplish reading the mapping file:

1. The **!kmap** command: `!kmap` *mapfile*
   This command can be typed into the prompt line and will cause the named file to be read and the mapping asserted immediately.

2. The `ReadKeymap` script function: `ReadKeymap(`*mapfile*`)`
   The function can be run from any script. In particular, this line can be added to a startup script (`.xicstart` or `.xicinit` file) to assert the mapping when *Xic* starts.

In either case, the *mapfile* is a path to a file as created with this command. If the path is not rooted, the file is searched for relative to the current directory, the user's home directory, and through the library search path, in that order. The commands can be re-run with different mapping files at any time, the current mapping state reflects the most recent mapping applied.

13.2.1 Key Mapping File

The file produced in this manner contains all key mapping and action translation tables used by *Xic*. Although it is not really recommended, this file can be customized by the user, with a text editor. The recommended way to alter a key sequence response is with a macro (see below), but there may be occasions where the mapping file should be changed to achieve a desired effect. Contact Whiteley Research for assistance.

The name of the file is `xic_keymap.`*hostname*, where *hostname* is the name of the machine. When *Xic* starts, it looks for a file of this name in the current directory, then in the user's home directory, then in the library search path. If a file is found, it is read and processed.

The first section of the mapping file contains a listing of the keysyms for the special keys mapped with this command (**Home**, **Page Down**, etc.). The keysym is a system-defined number assigned to that key. In Unix/Linux, keysyms are defined in the X include file `X11/keysymdef.h`.

The next section of the file consists of the definition of the "macro suppression" character (described below). This must be a printable ASCII character, surrounded by single quotes.

The third section contains a mapping table for keysyms to an internal code. The first column contains the keysyms; a numeric value in Windows, or the standard name under Unix/Linux. The second column is the internal code. The third column is a subcode used only for function keys.

The next section contains an action table which maps actions before the keypress code is sent to any internal command in *Xic*. In action tables, the first column is a code which is either an internal code, or the ASCII value of the keysym. This is operating system dependent. Under Unix/Linux, the keysym code for a printing character is simply the ASCII value of that character. The internal code is interpreted numerically as a value (hex) 0 – 1f. Ascii values are (hex) 20 – fe. Other values are taken as keysyms. Under Windows, the code is similar, but the Windows keysym is used. The Windows keysym differs in that (1) the upper-case alpha characters must be specified rather than lower case, (2) for punctuation which is the Shift of a numeric key, the numeric key should be given, (3) for punctuation which is not a Shift value for a numeric key, a special keysym for that key should be given.

The second column is the modifier state. The actions should be listed in order of most specific to least specific with regard to modifiers. A value "0" in this column indicates "don't care". Otherwise, the entry consists of one or more of “SHIFT”, “CTRL”, and “ALT”, separated by a minus sign ‘-’. The actual modifier should be listed, even if a Shift state is implied by the value in column one.

For example, to select the ‘!’ character, in Unix/Linux one has

```
'!'         SHIFT
```

in Windows, this would be

```
'1'         SHIFT
```

The third column in the action tables is the name of an action, which is defined internally in *Xic*. This is the action performed when the keypress combination specified in columns one and two is detected.

The next section in the file is an action table that specifies actions to perform after the present *Xic* command processes the keypress. Many of the *Xic* commands look for specific keys, and if that key is seen, further mapping is inhibited. If the keypress is not used by the command, it is available for translation by this final table.

The last section of the file contains a “<Buttons>” field which maps the functions of the mouse buttons under Unix/Linux. This allows the functions of the buttons within *Xic* to be permuted. However, the functions of the buttons with respect to the user interface, such as the mouse button used to engage user interface buttons and menu items, will not change. This field is ignored under Windows.

13.3 The Define Macro Button: Assign a Macro to a Key

Pressing the **Define Macro** button in the **Attributes Menu** allows the user to enter a macro. The generated macros are stored in a file named `.xicmacros` or `.xicmacros.`*ext*, where *ext* is the current technology file extension. In all cases, when a macro file is produced, which occurs after any new macro is defined, the file is written in the current directory as `.xicmacros`. The new file contains definitions for all current macros. The user can add the suffix and move the file to their home directory if desired.

When *Xic* starts, it looks for a macro file in the following sequence, and inputs the first one (and only one) found:

1. current directory using same extension as tech file.
2. home directory using same extension as tech file.
3. current directory with no extension.
4. home directory with no extension.

Under Windows, the home directory is obtained from environment variables, in particular the value of HOME.

Macros can be attached to any combination of a keyboard key and modifier key(s), with the exceptions of **Enter**, **Esc**, **Backspace**, and the “bare” modifier keys (**Shift**, **Ctrl**, etc). However, not all key combinations will work. For example, the expansion of keyboard menu accelerators occurs before macro expansion, so menu accelerator key combinations can not be used as macros. Also, key combinations that are intercepted by the window manager, which may include some combinations involving the **Alt** key, can not be used as macros. Otherwise, macro definitions have higher priority than most other functions in *Xic*.

Macros are *not* expanded when the prompt line is in text editing mode. The macro expansion can be suppressed for the next key combination by pressing the macro suppression character first. The macro suppression character is the backquote (`‘`). The macro suppression character is eliminated from the keypress buffer after the next key is typed. For example, suppose ‘`o`’ is mapped to something, but you want to enter a literal ‘`o`’ to trigger the **Open** command (“`op`”). One would type “`‘op`”, after which the keypress buffer would contain “`op`” triggering the **Open** command.

While recording a macro, button and key presses will not have the normal effect, but the events are stored in the macro and will have the normal effect when the macro is invoked. Button presses will open a menu in the normal way, and selected menu commands will become active, but subsequent events will be swallowed by the macro recorder. In most cases, one can send events to a pop-up by performing the actions, which won’t be carried out but will be recorded in the macro.

Note that while recording a macro, if a command is initiated that uses the prompt line, the macro string display will be overwritten. It will come back after the first event. However, if the command uses the prompt line for input, the macro definition will be terminated as if **Esc** was entered.

Pressing **Enter** while a drawing window has the keyboard focus terminates the macro definition. Pressing **Esc** terminates a macro without saving it. **Backspace** removes the last event when defining a macro. To enter **Esc**, **Backspace**, or **Enter** into the macro, use **Ctrl-Esc**, **Ctrl-Backspace**, **Ctrl-Enter**.

For example, Suppose we want to define **Ctrl-x** to “press” the **Expand** button. In the **Define Macro** command, press **Ctrl** and **x** and release. In response to the next prompt, select the **Expand** button in the **View Menu**, move the pointer into a drawing window, and press the **Enter** key. From then on, **Ctrl-x** will be equivalent to pressing the **Expand** button.

To undefine a macro, define it with a null definition for the body.

See the description of the “!!” interface in Chapt. 19 for information on using script functions in macros.

In the present GTK-based user interface, there is less need for macros due to the rich set of keyboard accelerators available.

13.3.1 Macro File Format

The `.xicmacros` file can contain any number of macro definitions. It is not expected that most users will have a need to work with this file directly, though the possibility exists. Each macro consists of a block of lines in the following form:

```
#macro
KeyDown(...  , NULL)
statements
#end
```

Lines that start with ‘`#`’ that are not script preprocessor keywords (see 18.8) are taken as comments. The body of the macro consists entirely of calls to four script functions `KeyDown`, `KeyUp`, `BtnDown`, and `BtnUp`. The first line must be a `KeyDown` command which specifies the character mapped to the macro. The widget string is the value “`NULL`”. The remaining lines contain calls to these functions, which simulate the button and key presses recorded in the macro.

The events processed while *Xic* is in use are recorded in the `xic_run.log` file (see 2.9.1). This file and other log files are stored in a temporary directory, which is deleted when *Xic* terminates normally. To access the `xic_run.log` file from *Xic*, press the **Log Files** button in the **Help Menu**, then select the `xic_run.log` entry in the resulting file selector, then press the green octagon on the file selector. Advanced users can cut/paste sequences of the commands into script files or macros.

13.4 The Set Attributes Button: Set Window Attributes

The **Set Attributes** button in the **Attributes Menu** brings up the **Window Attributes** panel. The panel contains controls which affect presentation attributes in all drawing windows. The panel contains five tabbed pages: **General**, **Selections**, **Phys Props**, **Terminals**, and **Labels**

The controls found in the **General** page are as follows.

Cursor
This menu provides a choice of cursors. These include the system default cursor (which is probably the same as the left arrow), cross cursor, left and right arrows. Under Windows, there is no right arrow, so an up arrow is used instead (but it is ugly and useless). Also, the default cross cursor for Windows 7 service pack 1 is huge and grotesque, but can be switched for a better looking cross cursor through the selections in the Windows **System Preferences** panel.

Use full-window cursor
When this check box is checked, the mouse cursor will be represented by horizontal and vertical lines which extend across the entire width and height of the drawing window containing the cursor. The lines intersect at the nearest snap point in the current window.

When not checked, the cursor is the normal small cross.

This tracks the state of the FullWinCursor variable.

Subcell visibility threshold (pixels)
This entry area specifies a pixel size threshold for display of expanded subcells. Subcells with height or width less than this threshold will not be displayed, or displayed as an unfilled bounding box, according to the setting of the **Subthreshold Boxes** menu button in the **Main Window** sub-menu of the **Attributes Menu**, and the **Attributes** menu of sub-windows. The value can be in the range 0–100. If 0, subcells will always be rendered when in expanded state, which can greatly increase drawing time when zoomed out. This setting applies to all drawing windows.

This entry tracks the setting of the CellThreshold variable.

Push context display illumination percent
When the **Push** command is active, and the “context” (the features surrounding the pushed-to subcell) is being displayed, the intensity of colors used to render the context is reduced. This visually differentiates objects in the current cell from those in the context. The percentage intensity of the context can be set from this input area. If set to 100, the context is rendered with the same coloring as the current cell.

This entry tracks the setting of the ContextDarkPcnt variable.

Pixels between pop-ups and prompt line
For windows that are automatically placed just above the prompt line, giving this entry a positive integer value will position these windows toward the top of the screen by that many pixels. This is useful when using “plasma” displays (such as Mac or KDE), where the shadow falls on the prompt line, which can be distracting. It might also be helpful if the window positioning is incorrect, which might occur with some window managers.

This tracks the state of the LowerWinOffset Variable.

The **Selections** page contains the following two check boxes.

Show origin of selected physical instances
When this check box is set, selected physical instances will have the cell origin marked with a cross.

This applies to the selection highlighting, as well as to the ghost rendition which is attached to the mouse pointer during a move or copy operation.

Showing the origin may seem trivial, but marking the origin requires a bit of overhead since it requires running a transformation and keeping track of an additional redisplay area since the origin may be outside of the cell bounding box. Thus, the default is to not show the mark.

This tracks the state of the MarkInstanceOrigin variable, and applies to all physical drawing windows.

Show centroids of selected physical objects

In mathematics, the centroid or geometric center of a two-dimensional region is the arithmetic mean of all the points in the shape. When this check box is set, selected objects will mark the centroid with a cross. This applies to the selection highlighting, as well as to the ghost rendition which is attached to the mouse pointer during a move or copy operation.

This tracks the state of the MarkObjectCentroid variable, and applies to all physical drawing windows.

The **Phys Props** page contains the following controls.

Erase behind physical properties text

When this check box is set, in windows where physical properties are being displayed, the area around the physical property text is erased, providing improved visibility.

This tracks the state of the EraseBehindProps variable.

Physical property text size (pixels)

This entry sets the height, in pixels, of the text used to render physical properties on-screen, when physical property text is being displayed.

This tracks the state of the PhysPropTextSize variable.

The **Terminals** page contains the following controls.

Erase behind physical terminals

This will cause the area under terminals in physical windows to be erased, to promote visibility. One can choose to not erase, to erase only under the cell's terminals, or to erase under all terminals. This tracks the setting of the EraseBehindTerms variable.

Terminal text pixel size

This sets the text height, in pixels, of the text associated with terminals in both physical and electrical windows. This tracks the setting of the TermTextSize variable. The default text height is 14 pixels.

Terminal mark size

This sets the pixel size of the mark used to indicate terminal locations in both physical and electrical windows. It tracks the value of the TermMarkSize variable. The default mark size is 10 pixels.

Finally, the **Labels** page contains controls related to label presentation.

Hidden label scope

By default, all labels participate in a protocol whereby clicking on the label with the **Shift** key held will "hide" the label, displaying a small box instead. **Shift**-clicking on the box will return to the display of the label text. This menu limits the labels which will participate in this protocol. The choices are **all labels** (the default), **all electrical labels**, **electrical property labels**, and **no labels**.

This menu tracks the setting of the LabelHiddenMode variable.

Default minimum label height

This sets the minimum label height, in microns, for new text labels. The actual initial height may be larger, depending on the zoom factor of the window, but it can not be smaller.

This tracks the setting of the LabelDefHeight variable.

Maximum displayed label length

This entry sets the maximum width, in default-sized character cells, of a displayed label. If the label exceeds this width, it is not shown, and a small box at the text origin is shown instead. The default is 256.

The "hidden" status of a property label can be toggled by clicking the text or box with button 1 with the **Shift** key held. See 7.9 for more information.

This entry tracks the setting of the LabelMaxLen variable.

Label optional displayed line limit

Label text strings may have embedded newline characters which cause them to be displayed on multiple lines. This setting, when set to a positive integer value, provides a limit on the number of lines that are actually displayed, in labels that respect this limit. Only the first N lines would actually appear in the display, where N is the given number. If N is zero, there is no limit.

Labels observe this limit only if an internal flag is set in the label. Presently, this is set internally for the labels associated with value and param properties. The user can apply the limit to any label by setting the `LIML` flag in the `XprpXform` pseudo-property.

The setting tracks the value of the LabelMaxLines variable.

13.5 The Connection Dots Button: Show Connections

The **Connection Dots** button in the **Attributes Menu** brings up the **Connection Points** dialog, which contains three "radio" buttons which specify how connection points are to be indicated in schematics shown in electrical mode windows. This appears in *Xic* only.

If **Don't show dots** is selected, there will not be any indication of connection points.

If the **Show dots normally** button is selected, a "dot" will be shown at ambiguous connections points. These are wire vertices common to two or more wires (except for common end vertices of two wires), non-endpoint wire vertices common with device or subcircuit terminals, and any point common to three or more terminals or wire vertices. This selection is the default.

A dot will also be shown if a cell connection point lies on an internal (non-endpoint) wire vertex. In instances, this marks the connection location, and also ensures that a dot will be shown at this location, as one likely would not appear otherwise due to the logic used.

If the **Show dot at every connection** button is selected, a dot will be shown at every intersection recognized as a connection by *Xic*. This can sometimes be useful for debugging connection problems in drawings.

The **Connection Points** choices track and set the state of the ShowDots variable. This variable can be set in the technology file or a startup file to initialize the connection point indication mode.

13.6 The Set Font Button: Set Window Fonts

The **Set Font** button in the **Attributes Menu** brings up the **Font Selection** panel, which allows selection of the fonts used in the graphical interface. A drop-down menu provides selection of the various font targets. Pressing the **Apply** button will immediately apply the selected font to all visible windows which use the font.

Although many of the fonts used in the graphical interface can be set from this panel, the main font, the one used for menu text and control labels, must be set externally (it must be available before the graphical interface is created). This is probably set by your desktop environment, and there should be a tool available for customization. One can also likely set various "themes" which alter the appearance of the window decorations.

This is not true in Windows. Perhaps there is a better way, but one can do the following: In the directory `c:\Documents and Settings\`*your_username*, create a file named `.gtkrc-2.0` containing lines similar to

```
style "win32-font" {
    font_name = "tahoma 12"
}
class "*" style "win32-font"
```

You may need a DOS or Cygwin window to create the file, as Windows Explorer cannot create a file starting with a period.

The **Dump Vector Font** button in the **Font Selection** panel will dump the vector font used for text labels in the drawing areas to a file. The user will be prompted for the name of a file to use, the default name is "`xic_font`". Any existing file with the same name will be backed up with a `.bak` extension. A font file with this name found along the library search path will be read on program startup, and will define the label font, overriding the internal font. The user can start by dumping the internal font, and tweak this to their taste. The format of the vector font file is discussed in C.1.

The drop-down font targets list contains the following entries:

Fixed Pitch Text Window Font
This sets the font used in pop-up multi-line text windows, such as the **Files Listing** and **Cells Listing**, where the names are formatted into columns.

Proportional Text Window Font
This sets the font used in pop-up multi-line text windows where text is not formatted, such as the info and error message pop-ups.

Fixed Pitch Drawing Window Font
This is the font used in the coordinate readout, the status line, layer table, and the prompt line. It is not the font used to render label text in the drawing windows, which is a vector font generated by other means.

Text Editor Font
This is the font used in the **Text Editor** pop-up.

HTML Viewer Proportional Font
This is the base font used for proportional text in the HTML viewer (help windows). If set, this will override the font set in the `.mozyrc` file, if any.

HTML Viewer Fixed Pitch Font
This is the base fixed-pitch font used by the HTML viewer. If set, this will override the font set in the `.mozyrc` file, if any.

The **Font** button in the **Options** menu of the text editor brings up a similar panel, as does the **Font** button in the **Options** menu of the help viewer.

These fonts can be set in the technology file, and are updated to the technology file when a **Save Tech** command is given.

13.7 The Set Color Button: Set Colors Panel

The **Set Color** button in the **Attributes Menu** brings up the **Color Selection** panel. The panel can also be displayed by clicking on a layer in the layer table or layer palette with button 3 while holding down the **Ctrl** key. The current layer will become the clicked-on layer, and its color will be loaded into the editor.

The panel contains controls that are manipulated to set the components of the color of the currently selected layer or other drawing attribute. If the **Print Control** panel (induced by the **Print** button in the **File Menu**) is visible, the color set will be used for rendering the plot, if the plot driver supports definable colors.

Along the top of the panel are three drop-down menus. The leftmost menu selects between **Electrical** and **Physical**, which for attributes whose color differs between modes, this specifies which color to display or set.

The second menu contains two or three choices: **Attributesa** and **Prompt Line**, and in the case where the first menu choice is **Electrical**, a thrid choice **Plot Marks** is available. The choice here will determine the content of the third menu. Each entry of the third menu represents a color that can be adjusted.

The third menu choices for the three choices in the second menu are described below.

Attributes
These are the colors used in the drawing windows. Most of these colors can be separately set while in electrical or physical mode.

Current Layer	Current layer color
Background	Drawing window background color
Coarse Grid	Color used for coarse grid lines
Fine Grid	Color used for fine grid lines
Ghosting	Color used for “sprites” attached to the mouse pointer
Highlighting	Color used for highlighting, such as for DRC errors
Selection Color 1	One of two alternating colors used for selections
Selection Color 2	One of two alternating colors used for selections
Terminals	Electrical terminals
Instance Boundary	Boundary color of unexpanded instance
Instance Name Text	Name text color in unexpanded instance
Instance Size Text	Size text color in unexpanded instance, physical mode only

Prompt Line
These are the colors used in the prompt line and status line.

Text	Normal prompt line text
Prompt Text	Text color used for prompting
Highlight Text	Text color used for hypertext references
Cursor	Text cursor color
Background	Normal background color
Edit Background	Background color while editing

Plot Marks

These are the colors of the plot point marks used in electrical mode to indicate a node or current being plotted by *WRspice*. The default colors are the same as the trace colors used by *WRspice* for plotting.

The entries are: **Plot Mark 1** to **Plot Mark 18**.

The rest of the panel consists of the stock GTK-2 color selection widgets. There are six up/down buttons which can adjust the red, green, and blue values, or the hue, saturation, and intensity values. To the left is a color wheel, with a triangle inside. One can drag the marked triangle vertex around the outer ring to set the color, and drag the small circle in the triangle to set the lightness. All widgets automatically track the current color setting.

There is also a palette containing several colors, and a rectangular color display area for the current and previous colors. Colors can be dragged between the palette locations and the current color location. The eye-dropper button allows setting the current color from a clicked-on screen object.

The **Colors** button brings up a listing of color names and RGB values. Clicking on a list entry will load that color into the color selector.

The **Apply** button must be pressed to actually transfer the new color to *Xic*. For layer colors, drag/drop of a color to the entries in the layer table or layer palette can be done as well. When changing layer colors, or screen attribute colors such as grid colors, the main window and similar (same display mode) sub-windows will be redrawn. The NoPhysRedraw variable, if set, will suppress automatic redraw of physical-mode windows.

When the **Print Control Panel** panel is visible, i.e., in hard copy mode, the colors set will be used in that mode only, and in the plots if the printer driver supports it.

13.8 The Set Fill Button: Fill Pattern Edit Panel

The **Set Fill** button in the **Attributes Menu** brings up the **Fill Pattern Editor** panel. The panel can also be displayed by clicking on a layer in the layer table or layer palette with button 3 while holding down the **Shift** key. The current layer will become the clicked-on layer, and its fill pattern will be loaded into the editor.

This panel initially displays an array of sample fill patterns. By pressing the **Pixel Editor** button, the pop-up will instead display a large pixel editor window, from which stipple patterns can be created and modified. The pixel editor is pre-loaded with the pattern of the current layer when the pop-up was invoked. When viewing the pixel editor, the **Show Stores** button reverts the display to the sample patterns.

Patterns are moved between the various boxes and layers in the layer table by drag/drop. Press and hold the left mouse button while the pointer is over the box or layer entry that is the source of a pattern, and release the button after moving the mouse pointer over the destination box or layer table entry.

There are also **Load** and **Apply** buttons that will load the pattern of the current layer into the editor, or set the pattern of the current layer from the **Sample**, respectively.

The **Sample** box, which is visible in both display modes, is the entry and exit point for the pixel editor. Dropping a pattern into the **Sample** area will load the pattern into the editor. The current pattern in the **Sample** area, which matches that currently in the editor, can be dragged and dropped onto a layer, or into one of the storage areas. When the **Pixel Editor** window is visible, it is also a drop receiver, equivalent to the **Sample** area.

When a pattern is applied to a layer by any means, the main window and any sub-windows showing the same display mode will be redrawn. The automatic redraw can be suppressed in physical mode by setting the **NoPhysRedraw** variable.

When displaying the pattern array, there are 18 pattern windows visible. The first two of these are immutable, containing empty and solid fills. The remaining 16 are registers, preset with default patterns. These patterns can be changed by dropping a new pattern into the display box.

There are 64 pattern registers available, in four pages. When the patterns are visible, the **Page** spin button can be used to cycle between the four sets of registers.

To summarize, the following transfers are possible with drag/drop:

- To the **Sample** box from any pattern register box, or the empty and solid boxes, or from any layer in the layer table or layer palette. Drag patterns into the **Sample** area to allow editing with the pixel editor. The **Pixel Editor** window is also a drop receiver equivalent to the **Sample** area.
- From the **Sample** box to any pattern register box, or to any layer in the layer table or layer palette. Drag patterns into layers to set the pattern used for that layer. A pattern dragged into a pattern register box will replace the existing pattern with the one from the source.
- From any pattern register box, or the solid and empty fill boxes, to any layer in the layer table or layer palette, or to the **Sample** box.
- To any pattern register box, from any layer from the layer table or layer palette, or the **Sample** box. Note that it is not possible to drag/drop patterns between pattern register boxes directly, one can drag from a register to the **Sample** box, then from the **Sample** box to another register box.
- From any layer in the layer table or layer palette to any of the default pattern boxes except solid and empty, or to the **Sample** box.
- To any layer in the layer table or layer palette, from any of the pattern register boxes, or the solid and empty boxes, or the **Sample** box.

If the **Print Control** panel (induced by the **Print** button in the **File Menu**) is visible, when a new fill pattern is applied to a layer, the new fill pattern will be used for rendering the plot, if the plot driver supports definable fill patterns.

Note that the new pattern set for a layer will not be visible in the drawing windows until they are next redrawn (press **Ctrl-r**, or click with button 2 near the center of the window to redraw the window).

The color used to display patterns in the pop-up is the color of the current layer. Initiating a drag from a layer in the layer table or layer palette will change the current layer (and hence the color) to that layer.

The 64 "default" fill patterns can be saved in an `xic_stipples` file in the current directory with the **Dump Defs** button, which is visible when the pattern array is being displayed. If this file is found in the library search path, it will be used to initialize the pattern registers when *Xic* starts. A system default `xic_stipples` file is provided in the startup directory.

When a pattern (including empty) is dropped on a layer, the **Outline**, **Fat**, and **Cut** buttons at the bottom of the pop-up set additional attributes relating to the pattern display.

When a pattern from a layer is dropped into the **Sample** box, the buttons will change state to that saved in the layer. The pattern registers, however, do not have attributes, so that the buttons remain unchanged when a pattern is dragged from a register box, except to gray the **Fat** button if the new pattern is not empty.

For empty fill, there are three available outline styles:

1. A thin solid line boundary.
2. A thin dashed line boundary.
3. A thick solid line boundary for Manhattan boxes and polygons, and a thin solid line boundary for other objects.

If the **Outline** button is not in the pressed state, the thin solid line boundary (style 1) will be used. If **Outline** is in the pressed state, and **Fat** is not in the pressed state, a thin dashed outline (style 2) will be used. If **Fat** is pressed, thick segments (style 3) will be used, but only for edges of boxes and Manhattan polygons. A thin solid outline will be used elsewhere.

If the **Cut** button is in the pressed state, boxes are rendered with thin lines along the diagonals, forming an X over the box. Polygons (even four-sided rectangular ones) and wires are drawn normally. The “cut” attribute is often used to signify vias.

If **Cut** and **Outline** are pressed with empty fill, but **Fat** is not pressed, the diagonal “cut” lines will be drawn as dashed, as for the outline.

When the pattern is not empty, the options are slightly different. If **Outline** is not pressed, objects will not be outlined. If **Outline** is pressed and **Fat** is not pressed, the patterned areas will have a thin solid outline. When **Fat** is pressed, boxes and Manhattan polygons will be shown with thick outlines. These are the only boundaries available with stippled fill. The **Cut** button will produce diagonals over boxes, as in the empty-fill case. If the pattern is solid, none of the attributes will be used, and the buttons are grayed.

When a pattern is dropped on a layer, the state of these buttons set the attributes for the layer. This applies whether the pattern source is the **Sample** box, or one of the pattern register boxes. In particular, to set up a desired empty-fill presentation for a layer, one would set the buttons, then drag the “pattern” from the empty-fill box to the layer to set.

In the pixel editor, the **NX x NY** spin buttons control the size of the map. Each coordinate can range from 2 to 32. The pixel editor window changes to accommodate the different aspect ratios, keeping the actual pixels square.

The three buttons just below operate on the pixel map.

Rot90

Pressing this button will rotate the pixel map by 90 degrees. Note that this swaps the Nx and Ny values.

X

This will flip the map right-to-left (mirroring about the Y axis).

Y

This will flip the map top-to-bottom (mirroring about the X axis).

The basic operation in the pixel editor is to toggle the state of a pixel by clicking on it with button 1, but more complex possibilities exist. A hold and drag will operate on all of the pixels enclosed in or intersecting the defined rectangle, which is ghost-drawn. The operations are indicated in the table below.

Button	Figure		Shift	Ctrl
1	solid box	toggle	set	unset
2	open box	toggle	set	unset
3	line	toggle	set	unset

If **Shift** or **Ctrl** is held down *before* button 1 is pressed, the action will be as for button 2. The three columns from the right indicate the state of the modifier keys on button release which produces the stated effect on the pixels. The **Ctrl** press overrides **Shift** if both are pressed.

Pressing the arrow keys while the pop-up has the keyboard focus permutes the pixel editor bitmap in the opposite direction of the arrow. This is valuable for allowing layers with similar patterns to show through one another.

Pressing the **Dismiss** button in the **Fill Pattern Editor** will retire the editor. This has the same effect as pressing the **Set Fill** menu button a second time.

13.9 The Edit Layers Button: Edit Layer Table

The **Edit Layers** button in the **Attributes Menu** brings up the **Layer Editor** pop-up, which contains buttons for adding and removing layers from the layer table, plus a drop-down menu of removed layers which can be added back. When the pop-up first appears, the text area will be blank, or will contain the name of the last removed layer. The text area can be edited to provide the name of a new layer to add, or the name of a removed layer can be selected in the drop-down menu. After pressing the **Add Layer** button and clicking in the layer table, the new layer will be added at the location of the layer entry clicked on. One can also click beyond the end of the listed layers to put the new layer above the existing layers. With the **Remove Layer** button pressed, layers clicked on will be removed from the layer table and added to the list in the drop-down menu.

When layers are removed, the geometry on the layer is not affected, however it will be invisible on-screen (after the first redraw) and to all commands, and the geometry will not be included if the cell is updated to disk.

Layers can also be added, removed, and renamed with the **!ltab** command.

13.10 The Edit Tech Params Button: Edit Tech Keywords

The **Edit Tech Params** button in the **Attributes Menu** brings up the **Tech Parameter Editor**. The editor can also be displayed by clicking on a layer in the layer table or layer palette with button 3 while holding down the **Shift** and **Ctrl** keys. The current layer will become the clicked-on layer, and it will become the target layer for the editor.

From the editor, many of the technology file keywords, mostly associated with layers, can have their specifications added, deleted, or edited. Keywords that are not adjustable in the editor have an alternate means of control, such as the separate panels for setting layer colors and fill patterns. After modification, the **Save Tech** button in the **Attributes Menu** can be used to generate a new technology file that incorporates the changes.

The present editor applies to all keywords. In earlier *Xic* releases, there were separate keyword editors in the **Extract** and **Convert** menus, as well as an editor in the **Attributes Menu**. The different keyword classes, previously available in separate editors, are now available by selecting one of the four tabs at the top of the editor window. Each of these keywords is normally applied to a layer in the technology file.

The editor is configured for the class of keywords selected, meaning that the pull-down menu contains entries for those keywords, and the listing details those keywords only. The classes, and keywords that can be manipulated, are listed below.

Layer

These are miscellaneous layer attributes, as described in A.6.1.

```
LppName
Description
Symbolic
Invisible
NoSelect
Invalid
NoMerge
CrossThick
WireActive
NoInstView
WireWidth
```

Extract

These are parameters that are used by the extraction system, but have relevance elsewhere, such as in the cross section viewer. These keywords are described in A.6.4.

```
Conductor
Routing
GroundPlane
GroundPlaneClear
Contact
Via
Dielectric
DarkField
```

Physical

These generally set a physical property of the layer material, primarily in support of the extraction system. These keywords are described in A.6.5.

```
Planarize
Thickness
Rho
Sigma
Rsh
EpsRel
Capacitance
```

```
Lambda
Tline
Antenna
```

Convert

These are mostly for establishing the mapping between *Xic* layers and GDSII layer and datatype numbers, as described in A.6.3.

```
StreamIn
StreamOut
NoDrcDatatype
```

In addition to the per-layer keywords, there is a **Global Attributes** menu that allows a few miscellaneous non-layer parameters to be manipulated. These were defined as technology file parameters in *Xic* releases earlier than 4.1.7. In present releases, these are simply variables. The editor provides a "front end" for setting the variables.

Global Attributes

```
BoxLineStyle
LayerReorderMode
NoPlanarize
AntennaTotal
SubstrateEps
SubstrateThickness
```

The editor is similar to the **Design Rule Editor** found in the **DRC Menu**. When the editor first appears, the keyword specifications for the current layer in the selected class are listed. The specifications appear as they would in the technology file. Changing the current layer will update the listing to the parameters for the new current layer. Selecting a different keyword class will update the display to show the keywords in the selected class. The user can add new keyword lines or modify existing lines as desired.

To add a keyword specification, one selects the desired keyword in the keywords menu of the editor. The available keywords are listed in the drop-down menu, and the set available depends on the class tab currently selected. After clicking on a keyword in the menu, the user will be asked to enter the associated text in the prompt line, if the keyword requires it. The keyword will be applied internally and appear in the listing if there are no errors. A status message will appear, indicating success, or providing an error message.

When adding a keyword, redundant and inconsistent keywords that are already in the list, such as a previous instance of the keyword, are removed. In other cases, a pop-up message will appear if inconsistent keywords are found.

Clicking on a line in the listing will select the line. The text for the selected line can be edited, or the line deleted, with the **Edit** and **Delete** buttons in the editor's **Edit** menu. The **Edit** menu also contains an **Undo** button, allowing the last operation to be undone.

13.11 The Main Window Button: Attributes sub-menu

The **Main Window** button in the **Attributes Menu** brings up a sub-menu which is identical to the **Attributes** menu in the sub-windows produced by the **Viewport** button in the **View Menu**. The menu contains attribute settings which apply to the main window. When a new sub-window appears, its attributes are inherited from the main window, but can be reset for the sub-window through its **Attributes** menu.

Main Window Sub-Menu			
Label	**Name**	**Pop-up**	**Function**
Freeze Display	`freez`	none	Suppress redisplay
Show Context in Push	`cntxt`	none	Show context in subedit
Show Phys Properties	`props`	none	Show physical properties
Show Labels	`labls`	none	Show labels
Label True Orient	`larot`	none	Show labels transformed
Show Cell Names	`cnams`	none	Show cell names
Cell Name true Orient	`cnrot`	none	Show cell names transformed
Don't Show Unexpanded	`nouxp`	none	Don't show unexpanded subcells
Objects Shown	`objs`	none	Object display control
Subthreshold Boxes	`tinyb`	none	Show outline of subthreshold cells
No Top Symbolic	`nosym`	none	Electrical only, don't show top cell as symbolic
Set Grid	`grid`	**Grid Setup**	Set grid parameters

13.11.1 The Freeze Display Button: Suppress Redisplay

When the **Freeze Display** button in the **Main Window** sub-menu of the **Attributes Menu** or the sub-window **Attributes** menu is active, no cell structure is drawn in the window, only the grid and the cell bounding box. This is for use when working on a large, complex design when it is not necessary to see the structure and it is inconvenient to wait for the display. When active, "FROZEN" appears in the upper left corner of the window.

In a frozen window, certain other highlighting features may appear. In particular rulers and the viewport location indicators as used by the **Show Location** function will be displayed. Also, the outlines of selected objects will appear in these windows.

Frozen sub-windows have an additional feature: frozen sub-windows display the viewport of the main window, the reverse of the **Show Location** function for sub-windows (which displays sub-window viewports in the main window).

This allows a useful trick for viewing huge cells. Suppose that one has a large design which takes a long time to render, and one wishes to examine a small part of this design (the approximate coordinates are known). One can employ the following procedure. Freeze the main window and read in the design. Bring up a sub-window by clicking twice outside of the cell boundary, so that the sub-window is empty. Freeze the sub-window, then press the **Home** key with the cursor in the sub-window to center and fully view the top cell boundary. Use the grid and/or rulers to determine the region of interest. Drag with button 3 to define a rectangle in the sub-window surrounding the region of interest, then click with button 3 in the center of the main window. Un-freeze the main window, and this region will be displayed. The region shown in the main window is shown with a dotted yellow outline in the frozen sub-window.

13.11.2 The Show Context in Push Button: Control Context Display

When the **Show Context in Push** button in the **Main Window** sub-menu of the **Attributes Menu** or the sub-window **Attributes** menu is active, the context is displayed in the window when the **Push** command in the **Cell Menu** is active. The context is the surrounding geometry in cells other than the instance of the cell that was "pushed" into.

13.11.3 The Show Phys Properties Button: Show Physical-Mode Properties

The **Show Phys Properties** button in the **Main Window** sub-menu of the **Attributes Menu** or the sub-window **Attributes** menu enables the display of object properties on-screen while the window is in physical mode. The property text is placed near the leftmost vertex with largest y value of a polygon, the leftmost end of a wire, or in the upper left corner of the object's bounding box. Properties of the cell itself are not displayed.

Properties can be assigned with the **Properties** command in the **Edit Menu**.

Outside of any command, clicking on a visible physical property string allows the property to be edited. The property string must be close to or overlap the object. If the string is too far away from the object (if there are a large number of properties, or the object size is small) it can not be selected in this way. The user is first prompted for new text, then for a new property number. The clicked-on property will be replaced with the new values. If **Ctrl-d** is pressed at any time while responding to the prompts, the property will be deleted, with no replacement.

Properties with property numbers 7000–7104 are not displayed. These numbers are reserved for internal use and should not be assigned.

The **Erase behind physical properties text** check box in the **Window Attributes** panel, or equivalently the EraseBehindProps variable can be set to erase around the property strings, enhancing visibility.

13.11.4 The Show Labels Button: Control Label Display

When the **Show Labels** button in the **Main Window** sub-menu of the **Attributes Menu** or the sub-window **Attributes** menu is active, labels will be displayed in the window, otherwise label rendering is suppressed. It is unlikely that it would be necessary to turn off the display of labels, unless a layout has so many labels that important features are obscured. Labels are shown in legible orientation by default, however if the **Label True Orient** button is active, labels will be shown with all transformations applied.

13.11.5 The Label True Orient Button: Set Label Orientation

The **Label True Orient** button in the **Main Window** sub-menu of the **Attributes Menu** or the sub-window **Attributes** menu is sensitive only when labels are being displayed (the **Show Labels** button is active). When active, labels will be shown in true orientation, i.e., all transformations are applied to the text before display. If not active, labels will always be shown in "legible" orientation.

13.11.6 The Show Cell Names Button: Display Cell Names

When the **Show Cell Names** button in the **Main Window** sub-menu of the **Attributes Menu** or the sub-window **Attributes** menu is active, cell names and other information are printed within the bounding box outline of unexpanded subcells. The text is shown in a legible orientation, but if the **Cell Name True Orient** button is set, the text will be transformed in the same way as the cell.

13.11.7 The Cell Name True Orient Button: Set Cell Name Orientation

The **Cell Name True Orient** button in the **Main Window** sub-menu of the **Attributes Menu** or the sub-window **Attributes** menu is sensitive when cell names are being displayed in unexpanded instances (the **Show Cell Names** button is active). When set, the name text is transformed in the same way as the subcell. When not set, the name text is always shown in a "legible" orientation. It is sometimes convenient to invoke this option, as one can see at a glance which subcells are rotated, mirrored, etc.

13.11.8 The Don't Show Unexpanded Button: Don't Show Unexpanded Subcells

Normally, unexpanded subcells are shown as a bounding box, containing a cell name label and perhaps other text. When the **Don't Show Unexpanded** button in the **Main Window** sub-menu of the **Attributes Menu** or the sub-window **Attributes** menu is active, unexpanded subcells will not be displayed at all, i.e., they will be invisible.

13.11.9 The Objects Shown Button: Object Display menu

This button in the **Main Window** sub-menu of the **Attributes Menu** and the **Attributes** menu of subwindows brings up a sub-menu containing three checkable entries: **Boxes**, **Polys**, and **Wires**. All three entries are checked by default. If unchecked, objects of that type will not be shown in the display in the corresponding window. The window will have to be redrawn to see the effect, the redraw is not automatic.

Display of labels and instances is controlled by other buttons in the same menu.

13.11.10 The Subthreshold Boxes Button: Outline Tiny Subcells

When the **Subthreshold Boxes** button in the **Main Window** sub-menu of the **Attributes Menu** or the sub-window **Attributes** menu is active, subcells with a displayed size smaller than a threshold will be shown in unexpanded form, as an unfilled box. Otherwise, the cell will not be shown at all.

The threshold pixel size can be adjusted with the **Subcell visibility threshold (pixels)** entry area in the **Main Window Attributes** panel, or equivalently by setting the CellThreshold variable.

13.11.11 The No Top Symbolic Button: Enforce Schematic View

This button will appear in the **Main Window** sub-menu of the **Attributes Menu** or the sub-window **Attributes** menu only when the window is displaying in electrical mode. Thus, it never appears in *XicII* or *Xiv* feature sets.

When set, the top-level cell will be displayed as a schematic, whether or not the top-cell has an active symbolic representation.

Unlike the **symbl** button in the electrical side menu, this does not change the internal state of the cell (thus triggering the "modified" flag), and applies only to the window where set. It is available in electrical sub-windows in physical mode, when the electrical side menu is hidden.

All editing capability is available, so it is possible to edit the schematic and symbolic views of the same cell simultaneously, in different windows.

13.11.12 The Set Grid Button: Set Grid Parameters

The **Set Grid** button in the **Main Window** sub-menu of the **Attributes Menu** or the sub-window **Attributes** menu brings up the **Grid Setup** panel. Pressing **Ctrl-g** when a drawing window has keyboard focus will also produce this panel. The panel provides control of the grid display in the associated drawing window. Separate grid styles are available for electrical and physical mode, and in the main drawing window when the **Print Control Panel** is visible.

The panel is divided into two pages: **Snapping** and **Style**. The **Snapping** page is shown when the panel first appears. It provides control of the grid spacing and snapping, plus is the main control point for the edge snapping feature to be described. The **Style** page provides control over the visual presentation of the grid.

In most cases, the grid is not actually changed unless/until the **Apply** button, at the bottom-left of the panel, is pressed. This is not true of the **Edge Snapping** group, or the **All Windows** controls in the **Style** page, which work immediately, but do not force a screen redraw.

The following controls appear in the **Snapping** page.

At the top of the panel are entries which control the grid spacing and snapping. There is a coarse grid, and a fine grid, that may be displayed in drawing windows and hard-copy output. The coarse grid is an integer multiple of fine grid increments, the multiplier can be set from the **Style** page. If a grid would be too fine, it is not shown.

The snap grid represents points where the cursor is allowed to reside. These are related to the fine grid interval, there can be an integer number of snap points per fine grid interval, or an integer number of fine grid lines between snap points.

Snap Spacing

The **Snap Spacing** entry area will set the spacing, in microns, between snap points. The mouse pointer is constrained to fall only on snap points when geometry is being created. If the MfgGrid parameter has been set in the technology file, the **Snap Spacing** value is constrained to be a multiple of this value. The MfgGrid value is printed below the text entry area if set, or the word "`unset`" is printed if not.

The **Snap Spacing** entry displays the actual snap spacing in microns. If a `MfgGrid` has been defined, and one enters a snap spacing that is not an integer multiple of this value, the actual snap spacing will "snap" to the closest multiple before use.

When assigning a `MfgGrid`, one must consider the internal resolution, and the `MfgGrid` should be representable in the resolution in use. For example, the default resolution is 1000 per micron, or 1nm. If one attempts to use a `MfgGrid` of 2.5nm, round-off error will occur. To support this

`MfgGrid`, a resolution of 2000 would be required. The resolution is set with the DatabaseResolution variable.

When the panel first appears, the initial keyboard focus is to the **Snap Spacing** entry. Thus, text typed will go to this entry. Pressing **Enter** when a text entry has the focus calls the **Apply** callback and sets focus to the **Dismiss** button, where another **Enter** press will dismiss the panel. So, to quickly change the grid spacing from the keyboard only, one can type

Ctrl-g (adjust number) **Enter Enter**

SnapPerGrid or **GridPerSnap**

To the right is another text entry, which accepts an integer in the range 1–10. The entry label indicates **SnapPerGrid** or **GridPerSnap**, depending on whether the **GridPerSnap** check box, below the text entry, is checked. This controls the placement of the fine grid lines in the window.

If the label indicates **SnapPerGrid**, then the fine grid is spaced at the given integer times the **Snap Spacing** value. For example, if the integer is 3, then a fine grid line (or dot) will be drawn at every third snap point. there will be three snap points per fine grid interval.

If the label indicates **GridPerSnap**, then the fine grid is spaced such that the integer will give the number of fine grid lines per snap interval. For example, it the integer is 3, fine grid lines (or dots) will appear at the snap points, as well as the 1/3 and 2/3 proportional distances between snap points.

Note that when the integer is 1, there is no difference between the two cases.

In electrical mode, the snap interval should be a multiple of one micron, to avoid connectivity errors due to numerical roundoff. However, this was not enforced in older releases of *Xic*. Presently, sub-micron snapping on tenth-micron intervals is accepted, but with a warning issued. This allows older files to be "repaired", i.e., objects moved to a one micron grid. This is recommended for files that require it. A sub-micron snapping interval should not be used otherwise, and will not be saved in the technology file produced with the **Save Tech** button in the **Attributes Menu**.

The **Edge Snapping** group appears below the grid snapping controls. This is different from grid snapping. The edge snapping will snap the cursor to the edge of a nearby object, which may or may not be off grid ("off grid" means that the coordinate is not on a multiple of the snap grid interval). The edge snapping is used in, and the controls apply to, physical mode only. In electrical mode, the cursor will always snap to and indicate when near a connection point.

When snapped in this manner to an edge, a small dotted box transiently appears around the mouse pointer. If snapped to vertex of an object, the box will have a double outline.

Snapping will apply to visible objects at any level of the cell hierarchy. The edges of unexpanded subcells will also be snapped to.

The group consists of an **Edge Snapping** menu, and four check boxes, as described below. These may be set independently in the main window and sub-windows. When a new sub-window appears, it will inherit the edge snapping settings from the main window, but these can then be changed in the sub-window if desired.

Unlike other controls in the **Grid Setup** panel, these controls operate immediately. They do not require pressing the **Apply** button.

Edge Snapping Menu

The menu has three choices, to set the scope of the edge snapping mode. If **DISABLED** is selected, then there is no edge snapping in the window. The default entry, **Enabled in some**

commands, enables the edge snapping in commands where it may be useful. These include the side menu commands with the following keywords:

```
arc
box
break
donut
erase
polyg
round
wire
xor
```

as well as the **Rulers** command in the **View Menu**.

The third choice is **Enabled always**, which provides the edge snapping at all times, in commands or outside of any command.

Allow off-grid edge snapping
When this check box is checked, the cursor will snap to nearby edges, whether or not they are on grid. When not checked, only snapping to on-grid locations is done. This is unchecked by default.

Include non-Manhattan edges
When checked, non-Manhattan (meaning not horizontal or vertical) edges will be snapped to. The snap points are the intersection of the snap grid and the edge, plus the endpoints (vertices). If not checked, non-Manhattan edges ar not snapped to. This is unchecked by default.

Include wire edges
If checked, the edges of wires are considered for edge snapping. If not checked, wire edges are ignored. This mode is enabled by default.

Include wire path
A wire definition consists of a sequence of vertices, with an implied line path connecting them. This path would appear as a line running along the center of a drawn wire. When this box is checked, the wire path is considered for "edge" snapping. By default, this box is unchecked, so that the wire path is not snapped to.

The **Style** page contains controls which alter the presentation of the grid visually.

Show
If the **Show** button is active, the grid will be visible when gradations are adequately large. Otherwise, the grid will not be visible in the window.

On Top
If the **On Top** button is active, the grid will be drawn last, after all geometry. Otherwise, it will be drawn first, in which case it is more likely to be obscured by the geometry.

Store and **Recall**
The **Store** and **Recall** menus allow a set of grid parameters to be saved in an internal register, to be recalled as needed. There are separate physical and electrical grid registers. One will automatically save to and recall from the register associated with the window display mode.

The grid registers save all per-window parameters that can be set from the **Grid Setup** panel. There are seven registers available, as indicated on the menu produced by the **Store** button. The **Recall** button produces a menu with two additional entries:

revert
Revert the **Grid Setup** panel to the current window settings.

last appl
Recall the last settings that were applied with the **Apply** button.

Grid registers are saved to the technology file when a technology file is written, and are loaded when a technology file is read.

See also the description of the **!rg** and **!sg** commands. These can be used to save and restore the grid from registers.

Axes group
Below the **Show** button, a radio button group is provided to set the presentation style of the axes in physical mode. The choices are **No Axes**, **Plain Axes**, and **Mark Origin**. The **Mark Origin** choice is the default. The **Plain Axes** choice does away with the small box at the origin, showing the axes as simple lines. The **No Axes** choice suppresses the axes entirely. In electrical mode, the axes are always suppressed.

Coarse Mult
To the right of the radio group is an integer entry for the coarse multiple. This is the number of fine grid lines or dots per coarse grid line or dot. Acceptable values are 1 through 50. When set to one, the coarse grid replaces the fine grid, which is shown with the coarse grid color. The default value is 5.

Line Style Editor
This group sets the line or dot style used to render the grid. There are three “radio buttons”, **Solid**, **Dots**, and **Textured** that set the basic grid style. Choosing **Solid** will cause the grid to use continuous lines. The **Dots** option will use a grid consisting of a small dot or cross at each grid point. When this selection is active, a **Cross Size** entry area appears. This can be set to values 0–6, indicating the number of pixels to light up around the central dot in the four compass directions. If zero, only the central pixel is lit, which can be difficult to see on high-resolution displays. The value 1 generally looks like a much brighter dot. Larger values will appear as a small cross.

If **Textured** is chosen, a user-specified patterned line will be used, and the line style editing areas become visible. The line style editor allows the user to specify the patterning of the lines used to form the grid. The upper window is a sample of the current line style. The lower window allows the user to set the line style by clicking.

The line pattern starts at the left set bit (blue area) and extends to the right of the display. The pattern is used to “tile” the line. The left part of the display is shown in gray to indicate that it is not part of the line style mask. Clicking in this window with button 1 will toggle the bit. Button 2 will clear the bit, and button 3 will set the bit. Multiple bits can be set or toggled by dragging. The line in the preview window will reflect changes in the pattern.

The sample window is a drag source for a piece of text giving the line style mask in `0x`*hhhh* (hexadecimal) notation. The mask is the integer being represented by the lower window, with set bits in blue. This may be useful for creating line styles for entry elsewhere.

All Windows
This control group appears only in the **Grid Setup** panel for the main drawing window. The two controls in the group are different from other controls in the panel in that they apply globally to all drawing windows. They also differ from other controls in that they operate immediately without the need to press the **Apply** button, however an explicit screen redraw is necessary to see the effect.

The **No coarse when fine invisible** check box applies in physical mode only. When the check box is not checked, as one zooms out, when the fine grid becomes too closely spaced it won't be shown, however the coarse grid will be shown, unless it too is too finely spaced. If the check box is checked, the coarse grid will not be shown by itself, it will be suppressed when the fine grid is suppressed. This tracks the state (set or unset) of the GridNoCoarseOnly variable.

The **Visibility Threshold** entry sets the minimum number of pixels between grid lines or dots. The grid will be suppressed if it would be smaller. This applies to all drawing windows, both physical and electrical. This tracks the setting of the GridThreshold variable.

Pressing the **Apply** button will actually save the new grid parameters in *Xic*, and redraw the window if something has changed. Changes in the **Edge Snapping** and **All Windows** groups do not need the **Apply** button, changes take effect immediately. All other controls require an **Apply** button press to assert the change, and changes will **not** be saved unless **Apply** is pressed.

Chapter 14

The Convert Menu: Data Input/Output, Format Conversion

In addition to the native cell-per-file format, *Xic* has interoperability with the archive file formats listed below. These file types can be read into *Xic* directly with the **Open** command, and generated with the **Save As** command. The **Convert Menu** provides for setting format-specific conversion parameters, and contains other conversion commands.

Under Unix/Linux, files are opened in 64-bit offset mode. This enables files larger than 2Gb to be processed.

Native *Xic* cells use a CIF-like ASCII format, with one cell per file. This is the default format used by *Xic*, but is not particularly efficient with respect to input/output speed and disk space.

In addition to the native cell-per-file format, *Xic* supports a number of archive formats, which can contain one or more cell descriptions.

GDSII

The GDSII (Stream) format is an industry-standard binary file format for cell hierarchies and libraries. *Xic* can read Format Release 3–7 files, and write either Format Release 7 or Format Release 3 (which is readable on systems supporting Format Release 3–7). GDSII files that have been compressed with the GNU `gzip` program or equivalent can be read directly, and similarly compressed GDSII output can be generated by *Xic*.

The GDSII directives absolute magnification, absolute angle, and absolute path width are not supported in *Xic*. If found in input, the values are taken as relative, and a warning is issued. These are not supported by other file formats in a portable way, and should be considered obsolete.

CGX

The CGX (Computer Graphics eXchange) format is a public-domain binary archive format developed by Whiteley Research Inc. Similar in structure to GDSII, the advantages are more efficient data representation for reduced file size and ease of parsing for faster read/write. Although presently available only in Whiteley Research products, it is anticipated that the format will eventually be supported by other vendors. CGX files that have been compressed with the GNU `gzip` program or equivalent can be read directly, and similarly compressed CGX output can be generated by *Xic*.

OASIS

The Open Artwork System Interchange Standard (OASIS) is a new standard for mask layout data being developed by the SEMI organization. This is a binary format which features more compact representation and thus smaller files than GDSII.

More information is available from `wrcad.com/oasis`.

The present status of OASIS support in *Xic* is complete:

1. *Xic* can read any spec-conforming OASIS file.
2. OASIS output from *Xic* is readable by any other spec-conforming tool.
3. Exceptions to the above are **bugs**, please report!

Although it is "not documented", *Xic* can directly read OASIS files that have been compressed with the `gzip` program or equivalent. Unlike for GDSII files, this is not really supported, and it is not possible to write gzipped OASIS output from *Xic*. It is preferable to use the compression provided in the OASIS format.

CIF

The CIF format, though a bit archaic, is still popular. *Xic* supports a number of selectable dialects and extensions.

If the input file is in CIF format, and symbol (cell) names are not provided (i.e., no symbol name extension is found), the generated symbol names will be "`Symbol`*N*", where *N* is the integer symbol number given in the CIF file.

In general, files produced with this *Xic* release are NOT compatible with pre-*Xic*-4.0 releases.

The native cell files and CIF now accept and generate arbitrarily long layer names. These are not compatible with traditional CIF, or with older *Xic* releases. Older *Xic* releases will fail to read native cell files, or CIF files, with non-traditional layer names. Traditional CIF layer names contain four characters or fewer.

Native cell files and CIF files from the 4.0 branch that use traditional CIF layer names should be backward compatible in this respect.

There is a new syntax used for electrical node property strings. This will, in general, prevent backwards compatibility of schematic files. If the Out32nodes variable is set, files written will use the old node syntax, with loss of some data that is not supported by the older syntax, but files will be readable by older *Xic* programs. The **Convert Menu** entry brings up a menu containing commands which perform explicit translations and other manipulations and diagnostics.

The table below lists the commands found in the **Convert Menu**, and gives the internal name and a brief description.

Convert Menu			
Label	**Name**	**Pop-up**	**Function**
Export Cell Data	`exprt`	**Export Control**	Create a cell data file
Import Cell Data	`imprt`	**Import Control**	Read a cell data file
Format Conversion	`convt`	**Format Conversion**	Direct file-to-file format conversions
Assemble Layout	`assem`	**Layout File Merge Tool**	Merge layout data
Compare Layouts	`diff`	**Compare Layouts**	Find differences between layouts
Cut and Export	`cut`	**Export Control**	Write out part of a layout
Text Editor	`txted`	**Text Editor**	Text edit cell file

The **Open** command in the **File** menu can be used directly to read files in the supported formats for editing. When a cell is written to disk, it is by default written in the format of origin, though a format change can be coerced in the **Save As** command by supplying a file extension. Thus, there are alternatives to using many of the commands in the **Convert Menu**.

During a conversion, a log file is written by the converters. This file contains a record of messages emitted during the conversion. If during a conversion an error or warning message is emitted, a file browsing window containing the log file will appear when the conversion is complete, though this can be suppressed by setting the NoPopUpLog variable. These messages also appear on the prompt line during the conversion. The file browser is a read-only version of the text editor window, and has a number of associated keyboard commands, including word searching. See 3.13.2 for a listing of these commands.

On GDSII and OASIS input, if there is no specified mapping for a given layer and datatype, an attempt is made to map to the existing *Xic* layers, and if that fails, a new layer is created.

When reading CIF, layer names are matched to those defined in the current technology in a case-insensitive mode. This differs from native and CGX file types, which use case-sensitive matching. Layers found in the file which do not match any in the technology are created, using default parameters.

14.1 Feature Availability Table

The rather complicated table below describes how various features apply to the input and output generation panels and functions.

Operation From	Scaling	Layer Filter	Cell Name Mapping	Windowing
Windows				
Export Control	W		w C,F,P	o F,W,C
Import Control	R	Y	r A,C,F,P	
Format Conversion	C	Y	r C,F,P	c W,C,F,E
Open Cell Hierarchy	R		r C,F,P	
create CGD	C	Y	r C,F,P	c W,C,F,E
Open/**Place**/drag-drop				
Script Functions				
Current Cell				
`Edit`				
`OpenCell`	R	Y	r A,C,F,P	
`Save`				
Layout File Format Conversion				
`FromArchive`	C	Y	r C,F,P	c W,C,F,E
`FromNative`	C	Y	r C,F,P	
Export Layout File				
`SaveCellAsNative`				
`Export`	W		w C,F,P	o F,W,C
`ToXIC`	W		w C,F,P	o F,W,C
`ToCG`	W		w C,F,P	o F,W,C
`ToCIF`	W		w C,F,P	o F,W,C
`ToGDS`	W		w C,F,P	o F,W,C
`ToGdsLibrary`	W		w C,F,P	o F
`ToOASIS`	W		w C,F,P	o F,W,C
`ToTxt`				
Cell Hierarchy Digest				
`OpenCellHierDigest`			r C,F,P	
`ChdLoadGeometry`		Y		
`ChdEdit`	s	Y		
`ChdOpenFlat`	s	Y		w,c
`ChdWrite`	s	Y		w,c,f,e
`ChdWriteSplit`		Y		
Assembly Stream				
`StreamSource`		Y	r C,P	
Trapezoid Lists and Layer Expressions				
`ChdGetZlist`	s	Y		w,c
Polymorphic Flat Database				
`ChdOpenOdb`	s	Y		w,c
`ChdOpenZdb`	s	Y		w,c
`ChdOpenZbdb`	s	Y		w

Notes:

1. There are three internal global scale factor registers, which are set in the various windows and with the `SetConvertScale` script function. One scale (R) is for reading, another (W) is for writing, and the third (C) is for format conversion. This indicator is shown in the **Scaling** column. The lower case 's' applies to script functions that take a local scale value.

2. The layer filtering and aliasing module is a group of controls that appear in the **Format Conversion** panel and elsewhere. These maintain the values in the LayerList variable and related. A 'Y'

in the **Layer Filter** column appears where layer filtering can apply.

3. The **Cell Name Mapping** module is a group of controls that allow cell name aliasing, case changing, etc. This module appears in the **Format Conversion** panel and elsewhere. The state for this module tracks two sets of variables, similar to *InToLower* and *OutToLower*, depending on whether the panel is controlling input or output. The code letters in the **Cell Name Mapping** are:

r or w	Reading or writing variables.
A	The auto-aliasing feature for cell name clashes is available.
C	Case conversion is available.
F	Alias files can be used.
P	A prefix and/or suffix can be added to cell names.

4. The **Format Conversion** panel and others contain a windowing module, containing controls for entering a rectangle, plus **Use Window**, **Clip**, **Flatten**, and empty cell filtering buttons. Internally, there are two global register sets for the state of these controls, one for output and one for format conversion (windowing is never used for input). The `SetConvertFlags` and `SetConvertArea` functions can also be used to set the flag states and the windowing area.

   The codes in the **Windowing** column are:

c or o	Conversion or output values.
W	Windowing is available.
C	Clipping to the window is possible.
F	Flattening is available.
E	Empty cell filtering is available.

   If flattening (F) is listed first, the other options are only available when flattening. The option letters are listed in lower case for script functions that take local values as arguments.

14.2 Cell Name Mapping

Releases of *Xic* prior to 3.0.5 allowed white space in cell names. However, some *Xic* features, such as selection of cell names in the **Cells Listing** panel will not work with cell names containing white space, and there are probably many other examples. Most basic operations will work, though the cell name containing white space will have to be quoted when given in the prompt area and elsewhere. The use of white space in cell names can lead to trouble and is discouraged.

In the present release, by default, white space is not permitted in cell names. When reading archive files, the cell name alias mechanism (described below) is used to convert white space characters found in cell names to underscore characters. Attempts to open a new cell with a name containing white space will fail. However, white space is allowed, as in older *Xic* releases, if the NoStrictCellnames variable is set.

There is provision for modifying cell names as archive files are read, written, or format converted. The **Import Control**, **Export Control**, and **Format Conversion** panels available from the **Convert Menu** each contain a cell name mapping module for controlling modification of cell names during their respective operations. This module contains the following controls:

Auto-Rename
This is a choice in the **Default when new cells conflict** menu in the **Setup** page of the **Import Control** panel. Selecting this item sets the state of the AutoRename variable. When set, cell

names that clash with the name of a cell in memory encountered when an archive file is being read into memory will be changed to avoid a clash.

This will apply to files read with the **102208** command and equivalent, in addition to files opened from the panels and through script functions.

Prefix and **Suffix** text entries

Text entered into these text areas will be added as a prefix or suffix to cells encountered when reading an archive file. A limited text substitution mechanism is available. In the **Format Conversion** and **Import Control** panels, these text areas track the InCellNamePrefix and InCellNameSuffix variables. In the **Export Control** panel, these text areas track the OutCellNamePrefix and OutCellNameSuffix variables.

This will apply to files read with the panels and through script functions only.

To Lower and **To Upper** check boxes

If set, **To Lower** will convert upper case cell names to lower case, and **To Upper** will convert lower case cell names to upper. Mixed case cell names are not affected. Case conversion is performed before any applied prefix/suffix. In the **Import Control** and **Format Conversion** panels, these buttons track the state of the InToLower and InToUpper variables. In the **Export Control** panel, these buttons track the state of the OutToLower and OutToUpper variables.

This will apply to files read with the panels and through script functions only.

Read Alias and **Write Alias** check boxes

These buttons control whether an alias file (see next section) is read before a file is processed, and updated after processing is complete. In the **Import Control** and **Format Conversion** panels, the buttons track the InUseAlias variable, and in the **Export Control** panel, the buttons track the OutUseAlias variable. Aliasing from the alias file is applied before any other name change.

This will apply to files read with the panels and through script functions only.

GDSII conformance

When writing GDSII files, cell names will be forced to conform to the GDSII specification. For format level 3, this limits the cell name length to 32 characters. The character set is limited to alpha-numerics plus '?', '_', and '$'. This action is automatic when writing GDSII files and can not be disabled.

Device Library name clashes

When reading any of the archive formats into memory, if a cell name is encountered which clashes with a library device name, that cell name is modified. A warning message is added to the conversion log file indicating the change.

14.3 Cell Name Alias File

When reading and writing archive files, an alias file may be used or created. This file controls the renaming of cells between *Xic* and the archive file. Use of the alias file is optional, and by default is neither created or used.

The InUseAlias variable, if set (with the **!set** command or equivalent buttons), enables utilization of the alias file when reading from an archive. Similarly, the OutUseAlias variable enables utilization of the alias file when writing to an archive. These variables have corresponding buttons in the panels found in the **Convert Menu**.

If the variable is simply set as a boolean, i.e., to no value, the alias file will be read before a read or write operation, and created or updated if necessary after the operation completes. If the variable is set to a word starting with '`r`' (case insensitive), then the alias file will be read before the operation and used during the operation (if it exists), but will not be created or updated after the operation completes. If the variable is set to a word starting with '`w`' or '`s`' (case insensitive), the alias file will not be read before an operation, but will be created or updated after the operation completes.

If enabled, after a read/write operation on an archive file, an alias file may be created, or updated if it already exists. This file will be created in the same directory as the archive file, where it must remain in order to be found. The name of the alias file is the same as that of the archive file, with "`.gz`" stripped (if present) and "`.alias`" appended.

The alias file consists of lines with two tokens: the first token is a cell name found in the archive file, and the second token is the name of the cell as known to *Xic*, which will be different from the first token (i.e., cell names that are unchanged do not appear). The file will be used, if it exists and the operation is enabled, to translate cell names to and from the archive format, as the file is written or read. The alias file will be written or updated, if necessary and the operation is enabled, after an operation that reads or writes an archive file. No file is produced unless a name was changed.

On reading or writing an archive file, a name will potentially change if any of the cell name aliasing features are enabled. This includes enforcement of the GDSII standard for cell names when writing GDSII. Any name change will be indicated in the log file. If a name changes, the alias file will be updated, if updating is enabled. The sense of the substitutions from the alias file is reversed when reading vs. writing.

It is not an error if no alias file exists.

When the alias file utilization is enabled, one should be aware that the alias file is controlling cell naming when converting to and from that file, since occasionally this can lead to confusion. The values in the alias file have precedence over other directives, such as case changes. For example, suppose that an archive file is created with case mapping applied. This will produce an alias file, if updating is enabled. If the case conversion is then turned off, and the write operation repeated to the same file name with alias file reading enabled, the cell names will *still* be case-converted, due to the alias file. Similarly, when reading the archive file produced, the cell names will be back-converted by the alias file. If the translations are no longer wanted, the switches controlling alias file usage should be turned off, or the alias file deleted.

Note that it is possible for the user to hand edit the alias file to produce an arbitrary cell name mapping. For example, it might be used to convert all cell names in a design to nondescriptive random strings before sending a design file to another site, to mask the function of the circuitry.

14.4 Layer Names

Xic follows the Virtuoso/OpenAccess concept of component layers, purposes, and layer-purpose pairs. Component layers are represented by a name and a number, and are abstract. Likewise, purposes are an abstraction represented by a name and a number. An actual *Xic* layer, which appears in the layer table, is a layer-purpose pair.

Every *Xic* layer has a component layer name and purpose. The name of an *Xic* layer is given or printed in the form

component_layer[`:`*purpose*]

If the purpose name is "drawing", then it is not printed or given explicitly. Otherwise, the purpose is separated from the component layer name by a colon (':') character. Note that when the purpose is drawing, the *Xic* layer name is simply the component layer name, so if the only purpose used is drawing, the distinction between component and *Xic* layer names vanishes.

Example *Xic* layer names:

```
m1
m1:pin
```

The first name corresponds to component layer name m1 and purpose drawing. The second example uses a purpose named "pin".

In *Xic*, layer names of both types, and purpose names, are always recognized and treated without case-sensitivity. There is no limit on the length of these names. Component layer and purpose names can contain alphanumeric characters plus dollar sign ('$') and underscore ('_').

All of the component layer and purpose names also have corresponding numbers. These may be assigned by the user, or assigned internally by *Xic*. *Xic* will maintain the associations, but the numbers are not used by *Xic*. They are, however, important for compatibility with other tools such as Virtuoso.

All *Xic* layers may be given an alias name. The layer will be recognized by this name, as well as its normal name. *Xic* layers may also contain a description string, presentation attributes such as color and fill pattern, and a host of other flags and properties for use within *Xic*.

When working with GDSII and other files that use a numeric layer/datatype combination to designate layers, the layer/datatype combinations can be mapped into arbitrary *Xic* layers using the mapping constructs described in 14.6. If no such mapping is found, a default name will be used. The default name strings apply to *Xic* layers that use the default drawing purpose.

When the layer and datatype numbers are in the range 0–255 the default name string takes the form of a four-byte upper case hexadecimal integer. The two left characters indicate the layer number, zero padded. Similarly, the two right characters represent the datatype number. For example, layer 33, datatype 15 has the name "210F".

Xic supports layer and datatype numbers in the range 0–65535. Although values larger than 255 are outside of the GDSII specification, they are compatible with the GDSII file format and are used as extensions in some vendor's products. To represent the case where either value is larger than 255, an eight digit hex number is used. This is analogous to the four character encoding, but each field uses four characters.

When providing a layer name of this type to *Xic*, an alternate "decimal" form can be used. This is "*layer*,*datatype*" where the two integers are separated by a comma (no space). Thus, "33,15" is an equivalent way to specify the layer name for the example above. This is a convenience for entering layer names into the input fields of files and graphical windows of *Xic*. Internally, the layer name is always stored as the hex name.

In some cases when working with layer/datatype combinations, one of the two fields can be a wildcard. In the hex format, the hex digits of the appropriate field can be set to "X". In the decimal representation, a single '−' replaces the appropriate digits. For example, "0FXX" and "15,−" equivalently specify layer number 15 and any datatype number.

14.5 Layer Filtering and Aliasing

The **Import Control** and **Format Conversion** panels have a common module for layer operations. There is provision for controlling which layers from an input archive file are read. The default action is to read all layers contained in the archive file, however this can be changed for physical data only with the **Layer list**, and the **Layers Only** and **Skip Layers** buttons. Layers can be mapped to other layer names with the layer alias list, when enabled by the **Use Layer Alias** button. The layer alias list can be edited with a pop-up editor.

The module contains the following controls:

Layer List text area

The **Layer List** can be set to a space-separated list of layer names. Each layer name is expected to match an effective layer name in the file being read. For file types such as GDSII that designate layers with layer/datatype integers, either the hex encoding or decimal form can be used, with wildcarding accepted. This text area tracks the value of the LayerList variable.

Layers Only check box

If this box is checked, only the layers listed in the **Layer List** will be read. The button tracks the state of the UseLayerList variable.

Skip Layers check box

This box can be checked if the **Layers Only** box is unchecked, and this also tracks the status of the **UseLayerList** variable. When checked, layers listed in the **Layer List** will be ignored in input.

Use Layer Aliases check box

When set, the current layer alias list will be applied to layers found in the file. This button tracks the state of the UseLayerAlias variable. The layer alias list tracks the value of the LayerAlias variable. Aliases are applied before the **Layer List** tests.

Edit Layer Aliases button

This button brings up a panel for editing the layer alias list. This amounts to setting or modifying the value of the Layer Alias variable.

The panel contains a listing of two columns: the left column for layer names, and the right column for the alias. There are three drop-down menus: **File**, **Edit**, and **Help**.

The **File** menu contains entries for saving the layer alias list to a disk file, and for reading in the entries from a disk file.

The **Edit** menu contains entries to add, delete, and edit individual aliases, and to select listing layer names in "decimal" form.

A row in the listing can be selected by clicking on it. The selected entry is acted on by the **Delete** and **Edit** commands.

The **New** command brings up a text input pop-up to solicit a name and alias pair (separated by space and/or an equal sign). Both entries must be valid layer names or encodings. Either entry can use the decimal or hex notation, or can be a CIF name, as appropriate for the type of file.

If the **Decimal Form** menu item is checked, the listing will use the decimal form for layer/datatype entries. Otherwise, the hex form will be displayed.

14.6 GDSII Layer Mapping

The GDSII file format does not use layer names. Instead, geometry can exist on a numbered layer and datatype. Typically, the layer number and datatype can be in the range 0–255, or 0–63 for some older versions of the GDSII specification. Here, the combination of a GDSII layer number and datatype is referred to as a "specification".

Although the GDSII file format documentation, which is maintained by Cadence Design Systems, Inc., specifies the 0–255 range for the current GDSII release, the file format uses 16-bit integers to store these values, and other vendors support layer and datatype numbers outside of this range. *Xic* release 2.5.67 and later can semi-transparently handle layer and datatype values in the range 0–65535.

Although this section refers to the GDSII file format, the same mapping logic applies when reading OASIS and CGX files, when a layer/datatype are given.

When reading a GDSII file, *Xic* will attempt to map specifications encountered into existing *Xic* layers. If that fails, a new *Xic* layer will be created. The GDSII mapping for *Xic* layers is generally assigned in the technology file using the StreamIn keyword (for reading) and StreamOut (for writing), or can be specified with the **Tech Parameter Editor** from the **Edit Tech Params** button in the **Attributes Menu**. This is the primary means by which GDSII specifications are interpreted as *Xic* layers, but this requires *a-priori* knowledge of the content of the GDSII file.

This section describes the process *Xic* uses to map an unknown specification encountered when reading GDSII input, where "unknown" means that no suitable mapping exists in the StreamIn lines of the present *Xic* layers.

Xic will first try to identify an existing *Xic* layer to map to the unknown specification. The first test is to look for an *output* mapping (as produced with a StreamOut line) that matches. If a match is found, an *input* mapping will be created. The behavior depends on the setting of the NoMapDatatypes variable, which reflects the state of the **Map all unmapped GDSII datatypes to same Xic layer** check box in the **Setup** page of the **Import Control** panel from the **Convert Menu**. When this variable is set (directly with the **!set** command, or by the button), the datatype will be ignored. The following pseudo-code illustrates the logic:

```
loop through existing Xic layers {
  if Xic layer has no GDSII input mapping {
    if Xic layer output mapping = GDSII layer {
      if NoMapDatatypes set
        (use this layer)
      else if output mapping datatype = GDSII datatype
        (use this layer)
    }
  }
}
```

Each layer/datatype specification has an equivalent hex code. If the layer and datatype are less than 256, the hex code is of the form *LLDD*, where the *L*s are hex digits, zero-padded, which represent the layer number, and the *D*s similarly represent the datatype. If either number is larger than 255, the format is *LLLLDDDD*, which has the same interpretation, e.g., the *L*s are a four-digit zero-padded hex integer representing the layer number. If the NoMapDatatypes variable is in effect, the datatype field (the *D*s) can instead be filled with 'X' characters.

The hex values are produced in upper case, but matching is case insensitive.

If no suitable output mapping is found, *Xic* will look for layer names or long names which match the hex encoding. If a layer is found with a name or long name matching (case-insensitive) the hex code for the specification, and that layer has no input mapping, an input mapping will be created. The following pseudo-code illustrates the logic:

```
if NoMapDatatypes set {
  if layer and datatype less than 256
    hex_code = hhXX
  else
    hex_code = hhhhXXXX
}
else {
  if layer and datatype less than 256
    hex_code = hhhh
  else
    hex_code = hhhhhhhh
}
loop through existing Xic layers {
  if Xic layer name or long name matches hex_code
    (use this layer)
}
```

If no existing layer is found that can be mapped to, a new layer will be created. If the hex code is four characters, the name of the new layer is the same as the hex code. If the hex code is eight characters, the new name is an internally-generated unique four-character name, and the long name is assigned the hex code. The layer name in this case is in the form "*L???*" where *???* is a sequential decimal zero-padded integer, starting with "000". This mapping is also used in the four-character hex code case, if the hex code conflicts with an existing layer name.

After the GDSII file has been read, newly created layers will appear in the layer menu (they are added above existing layers). The user can modify colors, fill patterns, and other attributes for these layers, and dump a new technology file with the **Save Tech** command.

14.7 The Export Cell Data Button: Export Control Panel

The **Export Cell Data** button in the **Convert Menu** brings up the **Export Control** panel. The top of the panel contains tabbed pages for **GDSII**, **OASIS**, **CIF**, **CGX**, and **Xic cell files**. This selects the current output format, and exposes controls and settings particular to the format. The lower part of the panel has two pages. The **Setup** page allows setting of various format-independent parameters and modes which are used globally when writing layout files. The **Write File** page provides a button with which export of cell data to a disk file can be initiated, as well as settings which apply during the write operation.

Below are descriptions of the controls in the file format pages selected by the upper tabs. Clicking the tab exposes the page containing controls appropriate for the format. The parameters set with this panel always apply when writing layout files of the types indicated. In particular, the settings apply when the commands in the **File Menu** are used to save design data, as well as when the **Write File** button in this panel is used.

14.7.1 GDSII Settings

GDSII version number, polygon/wire vertex limit

This option menu effectively sets the GdsOutLevel variable. This determines the release number given in the GDSII file, and also sets limits on the number of vertices allowed in polygon and wire objects included in the file. If an object in the database has too many vertices, it will be written to the file as multiple objects, which cover the same area. The default is GDSII format release 7, which allows up to 8000 polygon or path vertices. It may be necessary to use one of the format release 3 choices if the file is to be read by older software.

Skip layers without Xic to GDSII layer mapping

When this button is active, layers without a GDSII output mapping will be ignored when producing GDSII or OASIS output, though a warning will appear in the log file. Otherwise, this is an error which terminates the operation.

This mode can also be enabled by setting the boolean variable NoGdsMapOk with the **!set** command.

GDSII files can be gzip compressed. Such files are recognized automatically on input, and can be coerced as output by giving a "`.gz`" suffix to the file name.

Accept but truncate too-long strings

The GDSII and CGX formats use a 16-bit integer to store record size, limiting the size of records to 64KB. This prevents storage of stings longer than this. By default, an attempt to write such a string to a GDSII or CGX file will generate a fatal error, aborting the operation. If this check box is set, overrunning strings will be truncated to maximum possible length, and the operation will continue without error. Warnings will appear in the log file, however.

The check box tracks the state of the **GdsTruncateLongStrings** variable.

Unit Scale

This entry area contains a value that will multiply the default values of the "machine unit" and "user unit" parameters which are used in the GDSII file, and all coordinates in the file will be divided by this value. The default values for these parameters are

machine unit:	1e-6/*resolution*
user unit:	1.0/*resolution*

where *resolution* is the internal resolution, which defaults to 1000 per-micron, but can be changed with the DatabaseResolution variable. It is not likely that the user will need to set this, and unless the user understands the implications it is recommended that the default value (1.0) be used. This entry area is an interface to the GdsMunit variable.

14.7.2 OASIS Settings

Advanced

This button brings up the **Advanced OASIS Export Parameters** panel, which allows modification of the more obscure features employed when writing OASIS output (see 14.8).

Skip layers without Xic to GDSII layer mapping

This is equivalent to the corresponding button on the GDSII page.

Use compression

When active, created OASIS files will use compression. The contents of each CELL record and name table will be placed in a CBLOCK record, which should reduce file size. When not active, no compression will be used.

This mode can also be enabled by setting the boolean variable OasWriteCompressed with the **!set** command.

Use string tables

When active, all strings including cell names, properties, and labels are saved in indirection tables. Throughout the file, strings will be referenced by number. This should reduce file size. When not active, each string will be saved locally for each reference.

This mode can also be enabled by setting the boolean variable OasWriteNameTab with the **!set** command.

Find repetitions

When active, an attempt is made to identify identical objects that are placed in multiple locations, and use REPETITION records in OASIS output instead of writing multiple object records. This should reduce file size, but can be compute-intensive. When not active, no attempt is made to use REPETITION records, except for cell arrays.

See the description of the OasWriteRep variable (in E.21), which controls the use of REPETITION records in OASIS output. The **Advanced OASIS Export Parameters** panel contains an interface for effectively setting the OasWriteRep variable string. The **Find repetitions** button will set this variable to the current string, or unset the variable. With the default parameters, the string is empty.

Write crc checksum

When active, a cyclic-redundancy (CRC) checksum is added to OASIS output files (OASIS validation method 1). When not active, no checksum is added.

See the description of the OasWriteChecksum variable (in E.21), which controls the validation method in OASIS output. This variable can be set explicitly to use byte-sum checksum validation (OASIS validation method 2). The check box sets/unsets this variable as a boolean.

14.7.3 CIF Settings

Extension Flags

This drop-down menu provides access to a number of checkable buttons which correspond to flags which enable various CIF format extensions. There are two banks of flags, the bank displayed is initially determined by the state of the **Strip For Export** button in the **Export Control** panel, or equivalently the state of the StripForExport variable. The top entry of the menu indicates this state. Clicking this entry will switch the menu to display and control the other bank of flags. The default values for the flags in the **Strip For Export** inactive case are all set, so all extensions are turned on. The other bank has all flags unset, so by default no extensions will be used when Strip For Export is set. However, the status of any of the flags can be toggled with this menu.

The flag states track the value of the CifOutExtensions variable.

The format extensions enabled by these flags are described in B.3, CIF Format Extensions.

The lower section of the CIF page contains three option menus which control aspects of the syntax used when writing CIF files. The three selectable variations are the syntax used for the cell name extension, the interpretation of the "`L` *layer*`;`" syntax element, and the syntax used for the label extension. *Xic* can handle almost transparently any of these syntax variations, however third-party applications may require a specific variation.

The selections shown in the menus tracks the value encoded in the CifOutStyle variable. When this variable is unset, the defaults (the first choice in each menu) are used.

Last Seen

When a CIF file is read into memory, the style of the CIF file is saved internally. Pressing the **Last Seen** button will update the three style menus to these saved values, by setting or clearing the CifOutStyle variable.

CIF Cell Name Extensions

Cell names were not part of the original CIF syntax specification. Various extensions have been used to supply cell names in a CIF file. Each of these extensions consists of command following the "`DS ...;`" command, in the following forms:

cname_index	**Historic Name**	**Format**
0	IGS	`9` *cell_name*;
1	Stanford/NCA	(*cell_name*);
2	Icarus	(`9` *cell_name*);
3	Sif	(`Name:` *cell_name*);
4	none	no extension used

In *Xic*, any of the first four forms (indices 0–3) will be recognized equivalently when reading CIF input.

CIF Layer Specification

Layers are specified in CIF in a command with syntax

`L` *token*;

The the *token* can be interpreted in two ways; as the name of a layer, or as an index into a layer table. For the second interpretation, the token must of course be an integer.

layer_index	**Historic Name**	**Format**
0	none	`L` *layer_name*;
1	NCA	`L` *layer_index*;

Of these, the first entry is most common. *Xic* can handle both of these interpretations (see 14.9).

If the indexing is selected for layers, the index will be 1–based, and correspond to the layers, left to right, in the layer table, i.e., the leftmost (lowest) layer in the layer table is designated index 1.

CIF Label Extensions

Text labels were not part of the original CIF syntax specification, so that various extensions are used to pass label information.

label_index	**Historic Name**	**Format**
0	Xic	`94 <<`*label*`>>` *x y orient_code width height*;
1	KIC	`94` *label x y*;
2	NCA	`92` *label x y layer_index*;
3	Mextra	`94` *label x y layer_name*;
4	none	no labels used

Unlike other extensions, the first extension listed above is unique to *Xic*. If other formats are used, label size and orientation information will be lost. When reading CIF input, any of these forms will be accepted.

14.7.4 CGX Settings

This page contains an **Accept but truncate too-long strings** check box, with purpose and functionality that is identical to the check box in the **GDSII** page.

CGX files can be gzip compressed. Such files are recognized automatically on input, and can be coerced as output by giving a "`.gz`" suffix to the file name.

14.7.5 The Setup Page

The **Setup** page contains a number of check boxes which control various modes when writing output, as described below.

Don't convert invisible layers

There are separate check boxes that apply to physical and electrical modes. When active, only layers that are currently visible, as selected with button 2 in the layer table, will be written when writing output using this panel. This is the method by which certain layers can be eliminated from generated output. When this button button is not active, all *Xic* layers will be written.

This feature can also be enabled by setting the variable SkipInvisible with the **!set** command.

Strip For Export

When the **Strip For Export** button is active, converted output will contain physical data only, and will contain no *Xic* extensions. Further, the **Strip For Export** check box implicitly enables the same functionality as **Include library cell masters** (see below), so that the file will not contain unresolved library cell references. Additionally, parameterized cell and standard via sub-masters will be included in output, as if the corresponding check boxes were also checked. The **Strip For Export** box should be checked when creating a file for use in generating photomasks. Note that the electrical information can never be recovered from a stripped file.

This mode can also be enabled by setting the boolean variable StripForExport with the **!set** command. The variable tracks the state of the check box.

Include library cell masters

If the input file references library cells (which have the LIBRARY flag set) whose masters are not found in the file, then these masters are not written to output unless this box is checked. With the box not checked, the output file will require the referenced library be preesent and open when the file is read into *Xic* (the same requirement as the input file). With the box checked, the masters are written to the output file, which will therefor not need the library.

This tracks the state of the KeepLibMasters variable.

Include parameterized cell sub-masters

When this check box is checked, output saved to disk files will include sub-master cells. Ordinarily, sub-master cells are not included, as they will be re-created when the file is read. However, when exporting to a system that does not support the pcells in use, the sub-masters must be written if the file is to have any value. With the sub-masters present, the cells/instances will look like normal cell placements.

This applies when writing all output, **except** when using the **Save** and **Save As** buttons in the **File Menu**, and the equivalent text accelerators and including the prompts when exiting the program. It is also ignored when using the `Save` script function.

Xic native pcells are only supported in *Xic*. OpenAccess-based pcells might be supported by other systems, that is certainly the intent of the Ciranova PyCells. Even if another system supports the

OpenAccess PyCells, it may not have the logic to rebuild the pcells coming from a GDSII or other file source. In that case one will probably have to ship the OpenAccess library files.

The StripForExport variable and the equivalent check box will have the same effect when set.

The PCellKeepSubMasters variable tracks the state of this check box.

Include standard via cell sub-masters
When checked, standard via cell sub-masters are included in the output file. This will be required when sending output to another system, as this implementation is specific to *Xic*. An exception may be systems that share an OpenAccess database with *Xic*. If the cells are written to the OpenAccess database, the standard vias should translate properly, and be recognized by other tools (e.g., Virtuoso) that share the database.

This applies when writing all output, **except** when using the **Save** and **Save As** buttons in the **File Menu**, and the equivalent text accelerators and including the prompts when exiting the program. It is also ignored when using the `Save` script function.

The StripForExport variable and the equivalent check box will have the same effect when set.

The ViaKeepSubMasters variable tracks the state of this check box.

Consider ALL cells in current symbol table for output
When checked, all cells in the current symbol table, not just the hierarchy of the current cell, will be output as if they were part of the hierarchy. The usual filtering of library and sub-master cells is retained. The resulting file may have multiple top-level cells. This mirrors the state of the `OutAllCells` boolean varible.

Don't flatten standard vias, keep as instance at top level
This mode may apply when flattening a physical cell hierarchy. When set, instances of standard vias are retained as such, rather than being written as geometry. This check box tracks the state of the NoFlattenStdVias variable.

Don't flatten pcells, keep as instance at top level
This mode may apply when flattening a physical cell hierarchy. When set, instances of parameterized cells (pcells) are retained as such, rather than being written as geometry. This check box tracks the state of the NoFlattenPCells variable.

Ignore labels in subcells when flattening
When flattening a cell hierarchy, if this check box is checked labels found in subcells are ignored, meaning not reparented to the top level. Pre-existing labels at the top level are not affected. This is intended to prevent net name labels from subcells conflicting with net name labels defined at higher hierarchy levels. The check box state tracks the NoFlattenLabels variable.

Keep bad output (for debugging)
When generating an archive file and an error occurs, the archive file will normally be deleted. However, if this box is checked. the output file will be given a "`.BAD`" extension and retained. This file should be considered corrupt, but may be useful for diagnostics. This tracks the KeepBadArchive variable.

14.7.6 The Write File Page, Exporting Design Data

The **Write File** page is used to initiate writing of a design data file to disk. A number of options are available when writing a file with this panel. Unlike the settings described above, these settings apply only to files created with this panel.

The currently selected tab at the top of the panel specifies the output format to use. Details of the format selections are described below.

GDSII

This choice will create a GDSII (Stream) file of the current editing cell and its descendents. Upon pressing **Write File**, the name of the file for the GDSII output is requested from the user. The user can add a "`.gz`" extension, or remove the extension if already present, to control whether or not `gzip` compression is used. The GDSII layer numbers and datatypes are as given in the technology file.

Xic will ensure that cell names included in the GDSII file conform to the standard (upper and lower case, digits, '_', '$', '?' only, up to 32 long in GDSII Release 3).

All layers that are to be written to the GDSII file should have a GDSII output mapping specified. This can be added to the technology file with a text editor, or interactively with the **Edit Tech Params** button in the **Attributes Menu**. By default, a layer needed for output that does not have a mapping will terminate the operation. However, if the **Skip layers without Xic to GDSII layer mapping** check box in the **GDSII** page of the **Export Control** panel is checked, or equivalently the variable NoGdsMapOk is set (with the **!set** command), then such layers will be ignored (producing no output).

OASIS

This choice will create an OASIS file of the current editing cell and its descendents. Upon pressing **Write File**, he name of the file for the OASIS output is requested from the user. The layer numbers and datatypes are as given in the technology file. These are the same as for GDSII.

All layers that are to be written to the OASIS file should have a GDSII output mapping specified. This can be added to the technology file with a text editor, or interactively with the **Edit Tech Params** button in the **Attributes Menu**. By default, a layer needed for output that does not have a mapping will terminate the operation. However, if the **Skip layers without Xic to GDSII layer mapping** check box in the **OASIS** page of the **Export Control** panel is checked, or equivalently the variable NoGdsMapOk is set (with the **!set** command), then such layers will be ignored (producing no output).

CIF

With this choice, the current editing cell and its descendents will be written to a CIF file. Upon pressing **Write File**, the user is prompted for the name of the file for CIF output.

The extension syntax used for cell name specification and labels, and whether the layer directives use indexing or names, are settable with the CifOutStyle variable and/or the **CIF** page menus in the **Export Control** panel.

CGX

With this choice, the current editing cell and its descendents will be written to a CGX file. Upon pressing **Write File**, the user is prompted for the name of the file for CGX output. The user can add a "`.gz`" extension, or remove the extension if already present, to control whether or not `gzip` compression is used.

Xic Cell Files

This choice will unconditionally write to native-format files the hierarchy of the current editing cell. It can be used to transform a hierarchy input from a supported archive format file into *Xic* native format.

When **Write File** is pressed, the user is given the option of setting the directory which will receive the created files. If no directory is given, the files will be created in the current directory. While the prompt is in effect, a pop-up containing a tree listing of the directory hierarchy rooted in the current directory appears. The user can select a directory in the listing, or type the directory path on the prompt line. If a directory path is given and the final directory does not exist, it will be created, if possible. Pressing **Esc** will abort the operation.

After the cell writing is complete, a library file will be written in the current directory, given the name of the top-level cell suffixed with "`.lib`". This file will have references to each of the new files created, with the top-level cell name listed first, and the others listed in alphabetical order. This library may be placed in the search path to gain access to the new files through the library mechanism, in which case the directory containing the files need not be in the search path.

The following controls are found in the **Write File** page.

Cell Name Mapping
This group of controls manages the cell name aliasing feature. This does not apply to native cell file output.

Windowing and Flattening
A subset of the windowing operations is available. From this panel, windowing is only available when flattening.

Conversion Scale Factor
The **Conversion Scale Factor** provides an entry area where a scale factor to be applied during the write operation can be entered. Values of 0.001 through 1000.0 are acceptable. This will apply to output initiated from this panel only.

Write File
The write is actually initiated with the **Write File** button. The name of the output file will be prompted for on the prompt line.

Cell files can also be written to disk using the **Save** and **Save As** commands in the **File Menu**. However, if scaling or other options available in this panel are required, the file must be generated from this panel.

14.8 The Advanced OASIS Export Parameters Panel: Set OASIS Parameters

The **Advanced OASIS Export Parameters** panel is provided from the **Advanced** button in the **OASIS** setup page of various panels, including the **Export Control** panel, the **Format Conversion** panel and the **Layout File Merge Tool**, all from the **Convert Menu**. It allows modification of the more arcane parameters used when generating OASIS output.

Don't write trapezoid records
This check box sets and unsets the OasWriteNoTrapezoids variable. When set, no attempt is made to save three and four-sided polygons in more compact trapezoid records. Setting this variable will likely increase file size but reduce writing time.

Convert Wire to Box records when possible
This check box sets and unsets the OasWriteWireToBox variable. When set, single-segment Manhattan wires will be saved in more compact rectangle records. This may reduce file size, at the expense of slightly longer writing time and loss of object type integrity.

Convert rounded-end Wire records to Poly records

This check box sets and unsets the OasWriteRndWireToPoly variable. When set, rounded-end wires, which don't have native OASIS support and are normally converted to extended-end (Manhattan extension) wires, are instead converted to polygons. The polygons require more memory than the wires, but preserve exactly the geometric coverage of the original layout, as rendered in *Xic*.

Skip GCD check

This check box sets and unsets the OasWriteNoGCDcheck variable. When set, the OASIS writer will not attempt to divide out a common factor in vertex lists, which is done to reduce file size but can have significant computational overhead.

Use alternate modal sort algorithm

This check box sets and unsets the OasWriteUseFastSort variable. When set, an older, less effective sorting algorithm is used to sequence objects in output to make use of modality. Use of this algorithm may reduce writing time but will potentially increase file size.

Property masking

This menu controls the OasWritePrptyMask variable, which can be used to avoid writing certain, or all properties. This can reduce file size if properties are not needed. The description of this variable explains this feature in detail.

The remaining controls provide an interface for setting the text string for the OasWriteRep variable, which is used to control the repetition finder. Use of the repetition finder is enabled/disabled by the **Find repetitions** check box in the **OASIS** page of the **Export Control** panel. The present panel sets the parameters to use when the repetition finder is enabled.

The repetition finder is a system that will identify identical objects, and attempt to identify periodic sequences of these objects in one and two dimensions. This can have a huge effect on file size, at the expense of computational overhead. The controls on this panel can be used to fine-tune the algorithm for a particular data set, producing, e.g., the smallest file, or reducing writing time.

The description of the OasWroteRep variable provides detailed information about the parameters found in the property string, which has the form:

OasWriteRep: [*word*] [`d`] [`r`] [`m`=*N*] [`a`=*N*] [`x`=*N*] [`t`=*N*]

The **Restore Defaults** button will reset all controls to the default values. The **Objects** check boxes control which object types are processed for repetitions, as for the *word* in the string.

The **Run minimum** is the value passed for the `m` option. Pressing the **None** button on this line will instead give the `r` option, and gray out the run and array controls. The **Array minimum** provides the value for the `a` option. The **None** button on this line will emit "`a=0`" and disable the entry area. The **Max different objects** line corresponds to the `x` option. The **Max similar objects** line corresponds to the `t` option. If the **None** button in this line is pressed, "`t=0`" will be emitted and the entry area is grayed. Note that these do not emit if the text area contains the default value.

To actually enable repetitions, the OasWriteReps variable must be set. This can be set by hand with the **!set** command or equivalent, in which case the controls above will take the values found in the string. Setting the **Find repetitions** check box in the **OASIS** page of the **Export Control** panel will also set the variable, to a string created from the state of the controls. When the variable is set, the listing of set variables brought up with the **!set** command without arguments can ge used to monitor the property string as the various controls are changed.

14.9 The Import Cell Data Button: Import Control Panel

The **Import Cell Data** button in the **Convert Menu** brings up the **Import Control** panel. The panel has two pages, the first is used to set various parameters that apply globally when reading design data into the *Xic* internal database. The second page provides the means to initiate reading data from disk files into *Xic*.

14.9.1 The Setup Page

This page contains a number of entries which control various defaults and features that apply when reading data files. These settings always apply when reading, in particular during use of the commands in the **File Menu**, as well as through use of the **Read File** page of this panel.

PCell evaluation: Don't eval native

Setting this check box will prevent evaluation of native pcells when an instance is found while reading file input. If an archive file contains the sub-masters, it is more efficient to use them rather than recreate them through evaluation. Note that if sub-masters are not provided, the super-masters for the pcells must be available.

This check box tracks the state of the NoEvalNativePCells variable.

PCell evaluation: Eval OpenAccess

Setting this check box will cause *Xic* to attempt to evaluate OpenAccess pcells when instances are encountered when reading file input. By default, this is not done, as evaluation is likely to fail, and the exporter has probably included the sub-master cells in the archive.

This check box tracks the state of the EvalOaPCells variable.

Don't create new layers when reading, abort instead

By default, when reading an input file, layers are created if necessary to match layers found in the file. The new layers are appended to the layer table. If the source is GDSII or another format such as OASIS that provides layer and datatype numbers, the new layer name will be an encoding of these numbers (see 14.6).

If this box is checked, new layers will not be created, and encountering a layer in input that is not mappable into an existing *Xic* layer will be treated as a fatal error.

The boolean NoCreateLayer variable tracks the state of the check box.

Default when new cells conflict

This menu determines the default behavior when a cell from a file being read conflicts with the name of a cell already in memory. There are five choices: **Overwrite All**, **Overwrite Phys**, **Overwrite Elec**, **Overwrite None**, and **Auto Rename**. If **AutoRename** is selected, when a name clash with a cell in memory is detected, the cell name of the cell being read is automatically changed to avoid the clash. A suffix "$N" is added to the cell name, where *N* is a small integer, and a warning message is added to the log file. The **Merge Control** pop-up will never appear in this mode. For the other four choices, in graphical mode, when a conflict is detected, the **Merge Control** pop-up will appear, if enabled. The initial state of the pop-up will be determined by this menu, but the actions can be modified by the user on a per-cell basis. If the pop-up does not appear either because it has been suppressed or the program is running in non-graphical (server or batch) mode, the default action will be performed.

The default cell name conflict behavior can also be set with three boolean variables: AutoRename, NoOverwritePhys and NoOverwriteElec. If AutoRename is set (with the **!set** command or otherwise), the other two variables are ignored, and the auto-rename mode is enabled. If none of these variables is set, then the default action is **Overwrite All**.

When a cell is encountered while reading an archive file or native cell into memory with the same name as a cell already in memory, and we are overwriting cells in memory, the new cell will overwrite the existing cell in memory in most cases. The exception is for existing cells that were read through the library mechanism. These cells have the IMMUTABLE (read-only) and LIBRARY flags set.

The IMMUTABLE flag has no bearing on whether or not a cell can be overwritten in memory. The overwritten cell will no longer be IMMUTABLE. In releases prior to 3.0.11, IMMUTABLE cells would not be overwritten.

If the existing cell has the LIBRARY flag set, it will be overwritten, unless the NoOverwriteLibCells variable is set. A warning message will be included in the log file in this case, but the read will be successful, with the result being as if overwriting was not enabled. If overwritten, the cell will no longer have the LIBRARY flag set. In releases prior to 3.0.11, LIBRARY cells would always be overwritten, unless IMMUTABLE was also set, which is the default for library cells.

Don't prompt for overwrite instructions

In graphical mode, when a cell name clash with a cell already in memory is detected while reading a file, the **Merge Control** pop-up may appear. This can be used to change whether or not to overwrite the cell in memory on a per-cell and per-mode basis. When this button is active, the **Merge Control** pop-up will not appear, and the overwriting will use the default setting.

This state can also be enabled by setting the boolean variable NoAskOverwrite with the **!set** command.

Clip and merge overlapping boxes

When this button is on, boxes on the same layer are merged together, if possible, as files are being read into the database. Overlapping boxes are clipped and/or merged. This applies to box objects only, and not polygons (even rectangular ones) or wires, and applies only for physical mode data. Electrical mode boxes are never merged. This tracks the setting of the boolean variable MergeInput, which can (equivalently) be set with the **!set** command.

This mode applies when reading input from a layout file, and is separate and unrelated to the object merging as controlled from the **Editing Setup** panel from the **Edit Menu**. These settings have no effect when reading layout data.

However, on layers where the `NoMerge` technology file keyword is set, box (or any object) merging is inhibited, in all cases.

Skip testing for badly formed polygons

When set, the reentrancy test for polygons is skipped while an input file is being read into the database. The default behavior is to check each polygon for potentially troublesome geometry specification while the polygon is being created.

This mode can also be enabled by setting the boolean variable NoPolyCheck with the **!set** command.

Duplicate item handling

When reading data from a layout file, identical objects and subcells placed on top of one another are sometimes found. Although these generally cause no harm, this is almost always a layout error. This menu provides three choices of how to handle the situation. The default action is to print a warning in the log file, but import the duplicate objects into the database. The **Remove Duplicates** choice will also issue a warning, but will not add the duplicates to the database. The third choice suppresses checking for duplicates entirely.

This menu tracks the status of the DupCheckMode variable.

Skip testing for empty cells

When set, there is no checking for empty cells as an input file is being read into the database, and the pop-up that normally appears when a file is opened for editing or viewing if there are empty cells in the hierarchy is suppressed. An "empty cell" actually means that both physical and electrical cells of this name either don't exist in the hierarchy, or contain nothing. It is possible to check for empty cells at any time with the **!empties** command.

This mode can also be enabled by setting the boolean variable NoCheckEmpties with the **!set** command.

Map all unmapped GDSII datatypes to same Xic layer

This setting affects only the creation of new layers when a GDSII or OASIS file is read into the database. The default behavior is to create a separate new *Xic* layer for each GDSII layer/datatype encountered that is not mapped in the technology file. With the variable set, all datatypes on the new GDSII layer are mapped to the same (new) *Xic* layer.

This mode can also be enabled by setting the boolean variable NoMapDatatypes with the **!set** command.

How to resolve CIF layers

This is an option menu which specifies how *Xic* interprets layer directives in CIF files.

The layer directive has the syntax

L *token*;

If the *token* is an integer, it might indicate the name of a layer with the name being the same integer string, or it might be an index into the layer table. The choices in the menu enforce these two behaviors.

The default resolution method (**Try Both**) works as follows: The parser reads "L *token*;". If *token* matches an existing layer name (as string comparison), that layer is accepted. If there is no matching layer, and the *token* is an integer in the range of 1 through a maximum number, and there is no leading 0, the token is tested as an index. if a layer exists with that 1-based index, that layer is chosen. If the layer still has not been resolved, a new layer is created in the layer table, with the given (numerical) name.

The option menu gives two additional choices. The **By Name** choice will skip the index test. If the string match fails with all existing layers, a new layer will be created. If the **By Index** choice is selected, the layer tokens are assumed to be integers. The string match test is skipped. If the index test fails, an error is reported and the operation aborts. New layers are never created in this mode. The layer tokens must be positive integers with no leading zeros that have a corresponding layer table entry.

The CifLayerMode variable corresponds to this set of options, where its value of 0–2 corresponds to the three choices.

Don't flatten standard vias, keep as instance at top level

This mode may apply when flattening a physical cell hierarchy. When set, instances of standard vias are retained as such, rather than being written as geometry. This check box tracks the state of the NoFlattenStdVias variable.

Don't flatten pcells, keep as instance at top level

This mode may apply when flattening a physical cell hierarchy. When set, instances of parameterized cells (pcells) are retained as such, rather than being written as geometry. This check box tracks the state of the NoFlattenPCells variable.

Ignore labels in subcells when flattening

When flattening a cell hierarchy, if this check box is checked labels found in subcells are ignored, meaning not placed in the current cell. Labels defined in the current cell are not affected. This is intended to prevent net name labels from subcells conflicting with net name labels defined at higher hierarchy levels. The check box state tracks the NoFlattenLabels variable.

Skip reading text labels from physical archives

When set, text labels will not be read from a layout file when reading physical-mode data. It is not generally advisable to use this, as text labels, though not physical objects, should be assumed to be present for a purpose. However, this check box gives the user the flexibility to strip these out.

In *Xic*, text labels are included when the bounding box of a cell is computed. If a text label actually determines the boundary of a cell, the bounding box of the cell may report differently from other tools. The effective size of a text label is not well defined, and other tools will probably make different assumptions about font size, etc., or may not include text labels in bounding box computations.

The state of this check box tracks the status of the NoReadLabels variable.

14.9.2 The Read File Page

This page provides a button to initiate reading a design data file into *Xic*, and various controls which set modes which will apply while reading.

Merge Into Current mode

This menu provides the option of merging the contents of another cell into the current cell, possibly recursively. Only the content associated with the present display mode is affected, for example in Electrical mode, only the electrical cells will be affected, the physical cells are untouched. Part of the motivation for this mode is to facilitate separate development of electrical and physical designs, allowing them to be merged at a later time.

If **No Merge Into Current** is the current selection, then merging is turned off. Reading a cell of the same name as the current cell can either overwrite the current cell or the new cell can be ignored, depending on how name clashes are currently handled (as set in the **Setup** page of this panel).

If one of **Merge Cell Into Current** or **Merge Into Current Recursively** is selected, and the **Read File** button is pressed, the following operations will be performed. The user will be prompted for a file name. The user can respond with the name of a file, or the name of a cell in memory.

If the user passes a cell name found in memory, the contents of that cell will be duplicated and added to the current cell. This completes the command. Note that there is no difference between the **Merge Cell Into Current** and **Merge Into Current Recursively** modes in this case.

If the file name is found on disk, the file will be opened in a temporary symbol table. If a cell is found in the temporary symbol table that has the same name as the current cell, the contents of that cell will be merged into the current cell. Otherwise, the user will be prompted for a cell name. If the user enters a valid cell name, the contents of that cell will be duplicated into the current cell. If the menu is set to **Merge Cell Into Current**, the command is done. The temporary symbol table will be cleared.

If a recursive merge is selected, the hierarchy of the current cell is traversed. For each cell in the hierarchy, if a cell with the same name exists in the temporary symbol table, the contents of that cell will be duplicated into its counterpart under the current cell. Care is taken to handle the details of this recursive merge cleanly.

There is no undo capability for this command, so be sure to save a copy of the current cell hierarchy before merging, in case of trouble.

Cell Name Mapping
This group of controls manages the cell name aliasing feature. The **Auto-Rename** button found here has the same functionality as the **Auto Rename** selection in the cell name resolution option menu. This applies only when reading archive input files, and not native cell files. The prefix/suffix modifications are applied only in input initiated from this panel or script functions.

Layer Modification
The layer change module allows layer filtering and/or mapping to be applied during the read operation. This applies when reading physical data only.

Windowing and Flattening
Windowing is available only when flattening. When flattening, all information read from a file is mapped intto the top-level cell in memory. Users should realize that flat representation can require lots of memory.

Conversion Scale Factor
The **Conversion Scale Factor** provides an entry area where a scale factor to be applied during reading can be entered. Values of 0.001 through 1000.0 are acceptable. This will apply to input initiated from this panel only.

Read File
The **Read File** button will prompt the user for a file to read into *Xic*, in the manner of the **Open** command. However for archive files scaling, layer filtering, etc. may be applied to the cells read from the file through use of this panel and not via the **Open** command.

14.10 Windowing Control Module

The windowing module is available in the **Format Conversion** panel and elsewhere, though not all features are available in some contexts. The module controls whether windowing and/or flattening is done when layout data are being processed.

Windowing

The **Use Window** button controls whether or not a rectangular area is to be used. If this button is set, only the objects that intersect this area will appear in the output. For subcells, only the objects that appear within the window for some instance will be converted in the corresponding cell. The rectangular area can be set with the **Left**, **Bottom**, **Right**, and **Top** entry areas. These are coordinates, in microns, in the coordinate system of the top-level cell, *after* scaling is applied. Only geometry that overlaps the window area will be included in the file. However, when viewing the new file, geometry in subcells that also exist outside of the window area will be visible, unless the hierarchy is flattened.

If the **Clip to Window** button is active in addition to the **Use Window** button, objects will be clipped to the given window. Without clipping, the entire object is retained. With clipping, the objects will be clipped to the window given. Again, unless the hierarchy is also flattened, geometry in subcells that also exist outside of the window will be displayed when viewing the new file.

When clipping, wires that require clipping are converted to polygons.

There are eight registers available for saving bounding-box parameters. With the **S** (store) button, the current values in the four text entry areas that define the rectangle can be saved in one of the registers. With the **R** (recall) button, the saved parameters can be retrieved into the text entry areas. These registers are shared with other pop-ups that used windowing. The 0 register is used by the **Cut and Export** command to save the rectangle defined with the mouse, the other registers are not directly used by any command. The **Cut and Export** command can be used as a short-cut for entering rectangle data through user of register 0. Press **Cut and Export** (in the **Convert Menu**), drag in a drawing window to define a rectangle, then press **Esc** to abort the command. Then, recall register 0.

Flattening

If the **Flatten Hierarchy** button is active, the output file will be a flat representation, i.e., all geometry will appear in the top-level cell, which will have no subcells.

Empty Cell Filtering

Occasionally it is important or desirable to remove empty cells from output, particularly when layer filtering is employed. Layer filtering can produce large numbers of empty cells. A large number of empty cells will increase file size and may produce inefficiency in downstream processing operations. Thus, provision for removing empty cells is available from the **Empty Cell Filter** check box group.

Empty cell filtering is recursive, in that it eliminates empty cells, and cells that contain only instances of empty cells. There are two empty cell filtering operations available.

1. The **pre-filter** uses in-memory per-layer/per-cell statistics gathered during Cell Hierarchy Digest (CHD) creation to identify cells that should be excluded due to layer filtering. This has relatively low overhead. The CHD in use must have been created with **per-cell and per-layer counts** specified, or this filtering is skipped. If a CHD is implicitly created in processing, i.e., the user is not using a named CHD from the **Cell Hierarchy Digests** panel, then these counts will be saved automatically.

   This filtering operation is performed entirely in memory and is typically very fast. However, it identifies only cells that are made empty due to layer filtering.

2. The **post-filter** identifies empty cells by reading the source layout file. This can be rather time consuming, but applies whether or not layer filtering is being used, and will identify all empty cells.

The two check boxes separately enable each of these empty cell filtering operations. If one doesn't care about empty cells, neither box should be checked. If one is using layer filtering and just wants a quick pass to remove cells made empty due to layer filtering, **pre-filter** should be checked. If one wants to remove all empty cells, both **pre-filter** and **post-filter** should be checked. This will generally provide the fastest operation. If not using layer filtering, this will be equivalent to checking **post-filter** only. When using layer filtering, enabling both filters can be much faster that using post-filtering only.

14.11 The Format Conversion Button: Format Conversion Panel

The **Format Conversion** button in the **Convert Menu** brings up the **Format Conversion** panel, which is a front end to a number of direct conversion functions which translate an input file into output of another (or the same) format. These are direct conversions, i.e., the data are converted directly and do not enter the main *Xic* database. This means that there are relaxed memory limitations, so almost arbitrarily large files can be translated. It is also possible to perform scaling, data windowing or clipping, and hierarchy flattening while translating.

Conversions can also be performed by reading in a hierarchy and using the explicit output conversion in the **Export Control** panel.

14.11.1 File Format Selection

A drop-down menu at the top of the panel selects one of four types of input:

Layout File
: The source file is a normal layout file in one of the supported archive formats. The various input file formats are recognized automatically.

Cell Hierarchy Digest Name
: Input will be read through a Cell Hierarchy Digest, as listed in the **Cell Hierarchy Digests** panel.

Cell Hierarchy Digest File
: Input will be read through a Cell Hierarchy Digest found in a file on disk, as was generated from the **Save** button in the **Cell Hierarchy Digests** panel.

Native Cell Directory File
: Input will consist of native cell files found in a given directory. All cells found in the directory that do not have a "`.bak`" file extension or duplicate a device library name, regardless of any hierarchical relationship or lack thereof, will be translated and concatenated into an archive file.

When translating CIF files, or from native cell files using **Native Cell Directory**, four-character CIF-style layer names found in the input must be mapped to layer and datatype numbers when output is in GDSII or OASIS format. If the layer exists in the layer table and the GDSII `StreamOut` parameter has been set, that mapping will be used. The `StreamOut` parameter is normally set in the technology file, but can also be set from the **Tech Parameter Editor** from the **Edit Tech Params** button in the **Attributes Menu**. When not mapped via an existing layer in the layer table, if the CIF layer name is a four-digit hex number, it will be interpreted as "LLDD" to obtain the GDSII layer and datatype numbers. If not in this form, a new layer number and datatype will be internally generated, using the UnknownGdsLayerBase and UnknownGdsDatatype variables.

When using **Native Cell Directory**, the directory can contain an alias file (see 14.3) that can be used to map native cell names to new names in the output. This file must be named "`aliases.alias`", and is never generated by *Xic*. It must be prepared by hand or some other means if needed. Each line contains the native cell name followed by the name to use in output, separated by white space. The **Read Alias** check box in the **Format Conversion** panel, or (equivalently) the **InUseAlias** variable must be set in order for the alias file to have effect.

The output format is selected through the tabs arrayed below the **Input Source** buttons. Each tab, when selected, displays a page that may contain format-specific settings. These pages are very similar to corresponding pages in the **Export Control** panel, and the settings in the two panels track. The **Format Conversion** panel provides some additional choices and options, however. The differences are described below.

GDSII

The output format is GDSII. When the **Input Source** is set to **Layout File**, this page contains an **Input File Type** menu. This menu contains two choices: **archive** and **gds-text**. The latter choice enables back-conversion to GDSII of the ASCII representation previously generated from a GDSII file using the **ASCII Text** output format tab. The **archive** menu choice should be selected when reading normal layout data.

The header of a GDSII file optionally contains information about fonts, reference libraries, and other things. This information is saved in a file named "`gds_header_props`" in the same directory as the output files, when converting to native files only. The file is subsequently ignored by *Xic*, as this information is not used by *Xic*.

OASIS

The output format is OASIS.

CIF

The output format is CIF.

CGX

The output format is CGX.

When translating to CGX format, the multi-box capability of BOX records in CGX is not used. However, this feature is used when CGX files are written from memory. Thus, reading a hierarchy into *Xic* and writing out a CGX file will probably result in a smaller CGX file than using the direct conversion.

XIC Cell Files

The output will be written to a family of native-format cell files.

When the selected output format is **Xic Cell Files**, the input will be converted to a number of native cell files, one for each cell defined in the input. The same result can be obtained by reading the input file into the database with the **Open** command, and then using the **Export Control** panel to generate the *Xic* files.

ASCII Text

The output will be converted to an ASCII text representation of the input file format, for GDSII, OASIS, and CGX input. This may be useful for debugging problematic layout files. The ASCII text format produced for GDSII can be back-converted to GDSII through use of the **gds-text** selection in the **Input File Type** menu of the **GDSII** page. The ASCII representation of OASIS files can be back-converted to OASIS with tools available from Anuvad. The two check boxes that appear on this page apply when translating OASIS:

OASIS text: print offsets

This sets/unsets the state of the OasPrintOffset variable, and when active the first token of each printed record contains the offset in the file or containing CBLOCK record. When not active, offsets are not printed.

OASIS text: no line wrap

This sets/unsets the state of the OasPrintNoWrap variable, suppressing line breaking when active. In this case, each record will use a single (possibly very long) line. When not set, lines are broken and indented.

Note that the **Input Source** choice will affect the availability of output format tabs, in particular if other than **Layout File** is selected, the available tabs are **GDSII**, **OASIS**, **CIF**, **CGX**.

Below the output format selection tabs, there are two tabs which alter the lower half of the panel. The **Setup** page provides some format-independent settings. The **Convert File** page provides the button to actually start the process.

14.11.2 The Setup Page

This page contains a number of check boxes which apply during format conversion. These options are described below.

Strip For Export

When the **Strip For Export** button is active, converted output will contain physical data only, and all masters, and will contain no *Xic* extensions. The **Strip For Export** box should be checked when creating an export file for use in generating photomasks. Note that the electrical information can never be recovered from a stripped file. The check box tracks the state of the StripForExport variable.

This mode can also be enabled by setting the boolean variable StripForExport with the **!set** command. The variable tracks the state of the check box.

Include library cell masters

If the input file references library cells (which have the LIBRARY flag set) whose masters are not found in the file, then these masters are not written to output unless this box is checked. With the box not checked, the output file will require the referenced library be preesent and open when the file is read into *Xic* (the same requirement as the input file). With the box checked, the masters are written to the output file, which will therefor not need the library.

This tracks the state of the KeepLibMasters variable.

Include parameterized cell sub-masters

When this check box is checked, output saved to disk files will include sub-master cells. Ordinarily, sub-master cells are not included, as they will be re-created when the file is read. However, when exporting to a system that does not support the pcells in use, the sub-masters must be written if the file is to have any value. With the sub-masters present, the cells/instances will look like normal cell placements.

This applies when writing all output, **except** when using the **Save** and **Save As** buttons in the **File Menu**, and the equivalent text accelerators and including the prompts when exiting the program. It is also ignored when using the `Save` script function.

Xic native pcells are only supported in *Xic*. OpenAccess-based pcells might be supported by other systems, that is certainly the intent of the Ciranova PyCells. Even if another system supports the OpenAccess PyCells, it may not have the logic to rebuild the pcells coming from a GDSII or other file source. In that case one will probably have to ship the OpenAccess library files.

The StripForExport variable and the equivalent check box will have the same effect when set.

The PCellKeepSubMasters variable tracks the state of this check box.

Include standard via cell sub-masters

When checked, standard via cell sub-masters are included in the output file. This will be required when sending output to another system, as this implementation is specific to *Xic*. An exception may be systems that share an OpenAccess database with *Xic*. If the cells are written to the

OpenAccess database, the standard vias should translate properly, and be recognized by other tools (e.g., Virtuoso) that share the database.

This applies when writing all output, **except** when using the **Save** and **Save As** buttons in the **File Menu**, and the equivalent text accelerators and including the prompts when exiting the program. It is also ignored when using the `Save` script function.

The StripForExport variable and the equivalent check box will have the same effect when set.

The ViaKeepSubMasters variable tracks the state of this check box.

Don't flatten standard vias, keep as instance at top level
This mode may apply when flattening a physical cell hierarchy. When set, instances of standard vias are retained as such, rather than being written as geometry. This check box tracks the state of the NoFlattenStdVias variable.

Don't flatten pcells, keep as instance at top level
This mode may apply when flattening a physical cell hierarchy. When set, instances of parameterized cells (pcells) are retained as such, rather than being written as geometry. This check box tracks the state of the NoFlattenPCells variable.

Ignore labels in subcells when flattening
When flattening a cell hierarchy, if this check box is checked labels found in subcells are ignored, meaning not reparented to the top level. Pre-existing labels at the top level are not affected. This is intended to prevent net name labels from subcells conflicting with net name labels defined at higher hierarchy levels. The check box state tracks the NoFlattenLabels variable.

Skip reading text labels from physical archives
When set, text labels will not be read from a layout file when reading physical-mode data. It is not generally advisable to use this, as text labels, though not physical objects, should be assumed to be present for a purpose. However, this check box gives the user the flexibility to strip these out. This check box tracks the status of the NoReadLabels variable.

In *Xic*, text labels are included when the bounding box of a cell is computed. If a text label actually determines the boundary of a cell, the bounding box of the cell may report differently from other tools. The effective size of a text label is not well defined, and other tools will probably make different assumptions about font size, etc., or may not include text labels in bounding box computations.

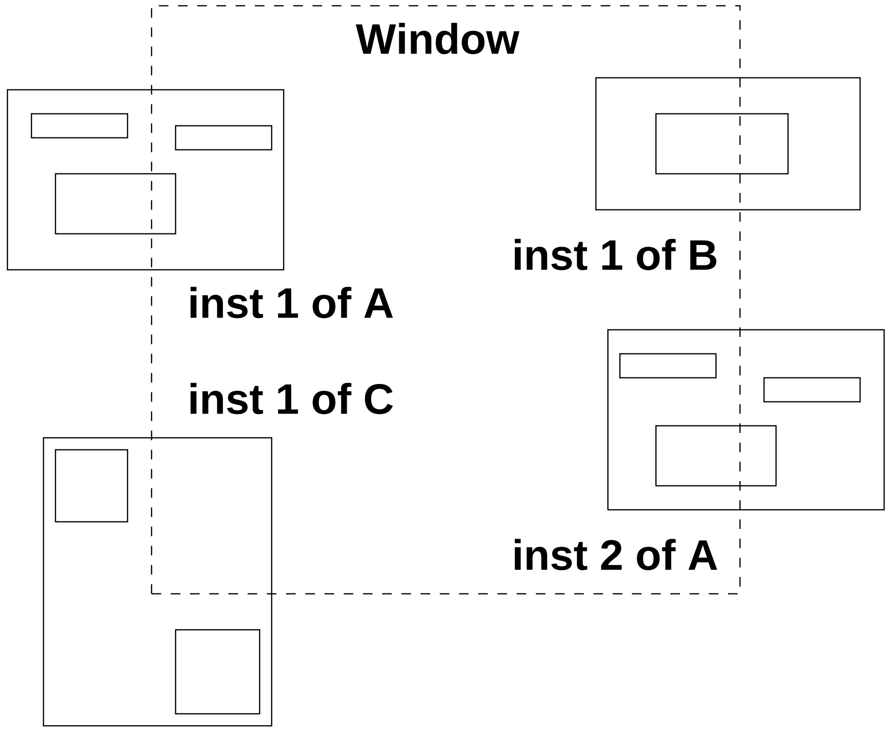
Keep bad output (for debugging)
When generating an archive file and an error occurs. the archive file will normally be deleted. However, if this box is checked, the output file will be given a "`.BAD`" extension and retained. This file should be considered corrupt, but may be useful for diagnostics. This tracks the state of the KeepBadArchive variable.

14.11.3 The Convert File Page

Layer Modification
The layer change module allows layer filtering and/or mapping to be applied during the conversion operation.

Cell Name Mapping
The **Cell Name Mapping** group of controls manages the cell name aliasing feature.

Figure 14.1: Illustration of windowing applied over subcell instances.



Windowing and Flattening

The windowing and flattening group can be used to set up area filtering or hierarchy flattening. These may not all be available for every input/output format permutation. For example, the windowing operations are not available when the input format is **Native Cell Directory**.

If windowing, flattening, or empty cell filtering is set, only physical data are converted, i.e., there will be no electrical data in the resulting file.

When windowing is in use and not flattening, an area filtering operation is applied to subcells. For each subcell, a bounding box is obtained that contains all of the intersection areas of instances of the subcell that overlap the window area, in the space of the subcell master. If there is no such overlap area, the subcell will not appear in output. Otherwise, only objects within the subcell that overlap this bounding box will appear in output. If clipping is enabled, the overlapping objects will be clipped to the bounding box boundary.

In figure 14.1, the two instances of A together “cover” all the objects shown in A. All of these objects will therefor appear in A in output as shown, whether or not clipping is enabled. They appear outside of the window boundary, illustrating that the window boundary is not absolute, unless flattening and clipping are employed.

In the single instance of B, the object shown straddles the window area and will therefor be included in output. If clipping is enabled, the object within B will be clipped to the window boundary. The single instance of C overlaps the window area, so will be included in output. However, since none of its objects appear within the area, the C subcell will be empty in output. Empty cells will be removed from output if the empty cell filtering option is set. This will add some computational overhead, and in most cases empty cells are “harmless”.

Conversion Scale Factor

For input file types that support scaling, the conversion scale factor entry area will be active. A scale factor of .001 – 1000.0 can be entered in this area, and will be applied during the translation. When scaling, only the physical (not electrical) data are scaled.

Convert

The translation is initiated with the **Convert** button. The user will be prompted for the name of the input file (or directory for **Native Cell Directory**, and then the name of the output file, or directory for native files.

When the input source is a CHD or saved CHD file, when the user is prompted for the CHD name or file name, the user can supply an optional second argument. This is the name of a cell in the CHD (including any aliasing applied when the CHD was created) that will be used as the root cell in output. If no cell name is provided, the top-cell configured in the CHD will be used. If no cell is configured, all cells referenced in the CHD will be converted.

If the input file contains multiple top-level cells, and no windowing, flattening, or empty cell filtering is employed, files are simply streamed through the converter and all cells are translated, using the specified parameters. If windowing or similar is employed, a temporary Cell Hierarchy Digest (CHD) is transiently produced in memory, which is used to perform the conversion. In this case, only the "default" top level cell hierarchy will be converted. This is the first cell in the file that is not used as a subcell by another cell defined in the file. Of course, if the input format choice is a CHD, and the CHD is configured with a top-level cell, that cell will be used.

14.11.4 Generating ASCII Output from Layout Data

The conversion of GDSII to "gds-text" is a diagnostic tool for converting the data in a (binary) GDSII file into a text form. Each record of the stream file is parsed and output generated in sequence. The text file can grow quite large, though a range specification can be given to limit the number of records printed. The text file is mainly used as a diagnostic for misbehaving GDSII files. It can be reconverted into a GDSII file, thus, the text representation is in effect another valid file format for layout data. This facility allows corrupted or otherwise problematic GDSII files to be repaired.

OASIS files converted to ASCII text use the same ASCII record format as `anuvad-0.8` from **SoftJin** (`http://www.softjin.com/html/anuvad.htm`), except for the separator lines that indicate the start of physical and electrical records. The `anuvad` tool set is free, and contains libraries and programs to convert between GDSII and OASIS formats, and to/from ASCII text representations of those formats. The boolean variable OasPrintNoWrap will suppress line wrapping when set, i.e., each record will occupy one possibly very long text line. The boolean variable OasPrintOffset will add file offsets to the output when set. These variables track the settings of the check boxes on the **ASCII text** output format tab page in the **Format Conversion** panel.

When converting to text format, the user will be prompted for an optional range specification string. If no string is given, the entire archive file will be written as text. The range specification string is expected to be in the following format.

[*start_offs*[`-`*end_offs*]] [`-r` *rec_count*] [`-c` *cell_count*]

The square brackets indicate optional terms. The meanings are:

start_offs

An integer, in decimal or "`0x`" hex format (a hex digit preceded by "`0x`"). The printing will begin at the first record with offset greater than or equal to this value.

end_offs
An integer in decimal or "`0x`" hex format. If this value is greater than *start_offs*, the last record printed is at most the one containing this offset. If given, this should appear after a '`-`' character following the *start_offs*, with no space.

rec_count
A positive integer, at most this many records will be printed.

cell_count
A non-negative integer, at most the records for this many cell definitions will be printed. If given as 0, the records from the *start_offs* to the next cell definition will be printed.

Records are printed from the beginning of the file, or the *start_offs* if given. Printing continues to the end of the file, or to the first of *end_offs*, *rec_count*, or *cell_count* if any of these have been given.

Back-conversion of the ASCII output into binary form is unlikely to succeed unless the whole file is written as ASCII.

14.12 The Assemble Button: Layout File Merge Tool Panel

The **Layout File Merge Tool**, brought up with the **Assemble** button in the **Convert Menu**, will extract cell hierarchies from one or more layout files, optionally perform some processing, then add the hierarchies to a single output file. It is essentially a graphical front-end for the **!assemble** command. The tool is intended to be highly flexible. Potential applications include building up reticles for mask generation or combining design output from different development groups into a single file.

Similar operations can be performed by use of reference cells.

The supported layout file formats for input and output are GDSII, CGX, OASIS, and CIF. The file format is specified when writing, and is determined automatically when reading. Any combination of these formats can be used for input and output.

The data read from the input files can be processed in various ways before writing to output. Some of these operations are sketched below.

- The layers can be filtered, to exclude certain layers, accept only certain layers, or to map certain layers to another layer. For GDSII and OASIS, the "layer" is actually a layer number/datatype number combination. Specification of a layer includes wildcarding of the layer or datatype number.
- The names of cells can be modified to add or replace a prefix and/or suffix.
- The data can be transformed by scaling, rotation, translation, and mirroring before placement in the output.
- The data can be filtered to objects that overlay a rectangular window, and may be clipped to the window.
- The hierarchy can be flattened before placement.
- Empty cells can be filtered out of the output.

Any combination of these processing operations can be specified.

The processed hierarchies from the specified input layout files are generally placed in a new top-level cell created in the output file. The user can choose the name of this cell, and if no name is given, no new top-level cell will be created, and the data from each input will simply be concatenated in the output.

During the merge, the tool remembers tha names of all cells seen, and will automatically change the names of cells that would clash in the combined file. Notification of the change will be written to the log file, which is produced during a run. The logfile is named "`assemble.log`" and is produced in the current directory.

14.12.1 Overview

Along the top of the **Merge Tool** are tabs which make visible separate pages for output and input. There will always be an output tab, and at least one source tab. At startup, there is a single source tab, labeled "Source 1". Each layout file from which files are to be extracted will have a source tab, and it is also possible to use the same archive file in different source pages if necessary. A new source page can be created with the **New Source** button in the **Options** menu, and an existing source page can be deleted with the **Remove Source** button in the same menu.

Each source page must be filled in with the appropriate entries before the merge run. We will return to a description of the fields in the source pages.

The left-most tab is labeled **Output**, and when selected will show a page for configuring the overall job output. The **Top-Level Cell Name** field may contain the name of a cell that will be created in the output file as a container for the cell hierarchies read from the sources. This will be the top-level cell in the output file. The name is arbitrary, but should conform to the standards of the output file format. If it should clash with another cell being written from a source, that file name will be modified to avoid the clash.

It is also possible to run a merge without entering a **Top-Level Cell Name**. In this case, the hierarchies extracted from the source archives are simply concatenated in the output file. The output file may then have multiple top-level cells. Any transformation information except scaling will be ignored, since transformations apply to the placement of the hierarchy in the container top-level cell.

The **Path to New Layout File** field is required; it specifies the output file. The format of the output file produced is determined by the format tab selected at the top of the output page.

The **Create layout File** button initiates to merge operation. It should be pressed when all relevant fields in the **Merge Tool** have been filled in. Depending upon the number and size of the files and hardware characteristics, the operation can take seconds to hours. When started, a progress monitor pop-up appears. This displays the number of bytes read and written, error and warning messages emitted, and a "working" indication. An abort button is also provided which can be used to terminate the operation.

The **Dismiss** button will exit the **Merge Tool** program. Unless the **Save** button in the **File** menu has been used, all entered information will be lost. The **Save** button can be used to save the current state of the **Merge Tool** to a file, which can be read later (with the **Recall** button) to configure the **Merge Tool** to the same state as was saved. The file format is that used as input to the **!assemble** command, and is described there. Note that files prepared by hand for use with the **!assemble** command can be loaded into the **Merge Tool** with the **Recall** button.

14.12.2 The Source Page

Each input file has at least one corresponding source page. Only one page is visible, and it occupies the main part of the **Merge Tool** display. The page displayed can be selected by clicking on the **Source** tabs just below the menu bar. The entries in the page identify the cells to extract and the processing to be performed.

The required **Path to Source** field contains the path to the associated layout file, and the file must be in one of the supported formats. This entry can also be the access name of a Cell Hierarchy Digest (CHD) in memory, or a path to a saved CHD file on disk. In either case, the CHD will then be used to access the content of the associated file.

14.12.3 Layer Filtering Module

The group of entries below the file path controls layer filtering and aliasing. These are optional and can be ignored if no layer manipulations are needed.

The module contains the following controls:

Layer List text area
The **Layer List** can be set to a space-separated list of layer names. Each layer name is expected to match an effective layer name in the file being read. For file types such as GDSII that designate layers with layer/datatype integers, either the hex encoding or decimal form can be used, with wildcarding accepted. The **Layer List** is ignored unless one of the following two check boxes is selected.

Layers Only check box
If this box is checked, only the layers listed in the **Layer List** will be read from the source.

Skip Layers check box
This box can be checked if the **Layers Only** box is unchecked. When checked, layers listed in the **Layer List** will be ignored in the source. All layers except those listed will be read.

Layer Aliases text area
This provides a means for converting layers found in input from the source to a different layer when written to output. The entered text contains zero of more space-separated text tokens in the form

oldname=*newname*

The *oldname* is a layer name consistent with the source format. For GDSII and OASIS, either hex or decimal encoding is accepted. The *newname* is the destination layer consistent with the output format. Again, gor GDSII and OASIS either the hex or decimal forms may be used. There should be no space between the names (or in the names) and the equal sign '=' separator.

14.12.4 Scaling

There are three different scaling entries which may apply. If there are no cells listed in the **Top-Level Cells** entry area, then none of the "per cell" settings (to be described) apply, and the value in the **Conversion Scale Factor** entry area will be used to scale all coordinates read from the source. The **Conversion Scale Factor** will be ignored if any cells are listed in the **Top-Level Cells** area, and scaling values will be obtained from the "per cell" entries.

The “per cell” entries allow scaling of the cell definitions written to output, and magnification of any instantiations created in a top-level cell in output.

Conversion Scale Factor numeric entry area
This provides a scale factor for cell data read from the source when no **Top-Level Cells** have been given. This value is ignored otherwise. This can range from .001 through 1000.0, and is applied to all coordinates of cells being read from the present source.

14.12.5 Cell Name Modification

This group allows systematic changes to the cell names read from the source layout file. If the source is a CHD, then the cell name modifications described here are performed after any cell modifications configured into the CHD.

Prefix and **Suffix** text entries
Text entered into these text areas will be added as a prefix or suffix to cell names encountered when reading the source file. The entries are string tokens, containing any alphanumeric characters plus ‘$’, ‘?’, ‘_’. String tokens given in this form will be prepended/appended to each cell name read from the source.

A limited text substitution mechanism is available. The string tokens can also have the form /*str*/*sub*/ which indicates a substitution. This causes the *str* if it appears as a prefix/suffix of a cell name to be replaced by *sub*. The *sub* can be empty (i.e., the form is /*str*//) which can be used to undo the previous addition of a prefix or suffix. Forms like //*sub*/ are equivalent to just giving *sub* as a string.

To Lower and **To Upper** check boxes
If set, **To Lower** will convert upper case cell names to lower case, and **To Upper** will convert lower case cell names to upper. Mixed case cell names are not affected. Case conversion is performed before any applied prefix/suffix.

14.12.6 Top-Level Cells List

Each source may have one or more top-level cells specified. If no top-level cells are specified the default operation will be as follows. If the source is a layout file, the entire file will be streamed into the output. If the source is a CHD, the cell hierarchy of the CHD’s default cell will be streamed into the output. If not explicitly configured, this will be the first top-level cell in the file referenced by the CHD.

The top-level cell names are names of cells in the source. If the source is a CHD with cell name modification, the names must include the modification. These cells, and possibly their hierarchies, will be used in output. Note that the cells listed are not necessarily top-level in the source, any cell in the source file can be listed.

Cells are added to the list by use of the **New Toplevel** button in the **Options** menu. Note that the user must generally know the names of the cells in the source to be extracted. If an empty cell name is given at the prompt, the text “`<default>`” will appear in the listing, which will correspond to the default cell of a CHD source or the first top-level cell found in a source file. Thus, it is possible to access the “top” cell in a source without knowing its name. Giving a cell makes available the “per cell” operations in the **Basic** and **Advanced** pages to the right of the listing.

One can use the **Contents** list in the **Cell Hierarchy Digests** listing to list the cell names, if the source has a corresponding CHD. Cell names can be dragged directly from the listing panel and dropped in the **Top-Level Cells** list, bypassing the need to use the **New Toplevel** menu button.

Clicking on a cell name in the list will select it, enabling additional "per-cell" entries which apply to this cell and its hierarchy. The selected cell name can be deleted from the list with the **Remove Toplevel** button in the **Options** menu.

14.12.7 Basic Transformations

If a cell name is selected in the **Top-Level Cells** listing, the entries in the **Basic** tab page become enabled. One may have to click on the "Basic" tab to display the entries. These control the transformation of the selected cell when instantiated in the top-level cell in the output file. If there is no top-level cell name given, these entries will be ignored.

Placement Name entry
: The **Placement Name** field can be filled in with a new name. The selected cell will be saved under this name, rather than its real name, in output. Any name modifications in force will be applied to this name.

Basic transformation entries
: This tab page contains entries that control the transformation of the selected cell when instantiated. The **Placement X,Y** entries set the translation coordinates in microns. The origin of the selected cell will be mapped to this location in the output top-level cell. Additionally, the cell instance can be rotated, mirrored, or magnified. The **Rotation Angle** menu provides rotation angle choices: multiples of 45 degrees. The **Mirror-Y** button will invert the Y-coordinates before rotation. The **Magnification** entry can change the scaling of the instantiation.

14.12.8 Advanced Operations

When a cell name is selected in the **Top-Level Cells** listing, the entries in the **Advanced** tab page become enabled. These operations apply to the cell data read from the source file for the selected cell, allowing windowing, flattening, and other operations. These are similar to the windowing operations provided in the **Format Conversion** panel.

Use Window check box
: Windowing operations are enabled by setting the **Use Window** check box. A window is a rectangular area in the selected cell, which is specified (in microns) with the four numerical entry boxes.

  With windowing enabled, only objects and subcells that have nonzero overlap with the window will be written to output. In subcells, only objects that overlap the window in the context of some instance will be included. Thus, only the objects in the file needed to represent the window area of the selected cell to all depths below the selected cell will be read from the source.

Clip check box
: If in addition the **Clip** check box is set, the objects will be clipped to the window. This includes objects in subcells. Note that this does not guarantee that geometry will not appear outside of the window, since instance geometry may appear anywhere.

Flatten check box

The **Flatten** button will flatten the hierarchy under the selected cell. Flattening can be applied with or without windowing. Along with windowing and clipping, no geometry will extend outside of the window area.

Empty Cell Filter

The **pre-filt** and **post-filt** check boxes enable the two stages of empty cell filtering, as described for the **Format Conversion** panel in 14.10.

Scale Factor entry

The coordinates in the cell and its hierarchy will be scaled by this factor in output. This is done logically before any windowing operations.

No Hierarchy check box

If checked, only the cell, and not its subcell hierarchy, will be included in output. This can lead to unresolved references in the output file.

14.12.9 Merge Tool Menus

The **Merge Tool** provides three drop-down menus in the menu bar at the top of the interactive display: **File**, **Options**, and **Help**. The **File** menu contains entries related to input/output, and **Options** contains entries for modification of program operation. The **Help** menu provides access to documentation. This section describes the entries of each menu in detail.

Some of the menu entries have keyboard accelerators, which are listed in the menu. Pressing the accelerator key combination has the same effect as pressing the menu button, without the need to display the menu.

14.12.10 The File Menu

The file menu contains command buttons that deal generally with input/output.

File Select

The **File Select** button brings up a **File Selection** panel. This enables the file hierarchy on the user's computer to be searched for files. Selecting a file by double clicking a name of pressing the green octagon "Go" button will enter the full file path into the **Path to Source** entry of the current **Source** page.

Save

The **Save** button will save the current **Merge Tool** configuration in a file. The file format is as described for the **!assemble** command in 19.2.3. This includes all of the filled-in entries of all pages currently recorded in the tool. This file can be subsequently read to reset the **Merge Tool** to the saved status. The generated file is in a simple ASCII format that can be generated by third-party scripts, etc., by the advanced user.

Pressing the **Save** button will pop-up a small dialog asking for a name for the file. This name can be anything, but it is recommended that a standard extension such as "`.sav`" be used to make these files easily recognized. Pressing the **Save State** button on the dialog will generate and save the state file.

Recall

The **Recall** button will read a file previously saved with the **Save** button, and reconfigure the **Merge Tool** to the state saved in the file.

Pressing the **Recall** button will pop up a small dialog asking for the name of the file. This can be entered directly, or the **File Selection** panel (from the **File Select** button) can be used to locate the file. Once located, the name of the file can be dragged from the **File Selection** panel and dropped in the dialog.

Pressing the **Recall State** button in the dialog will reconfigure the **Merge Tool** to the state found in the file. All entries in the tool should be as saved.

14.12.11 The Options Menu

The **Options** menu contains buttons that enable making certain entries into the **Merge Tool** forms, and otherwise induce changes in configuration.

Reset
Pressing this button will reset the configuration of the **Merge Tool** to the startup (empty) configuration. All existing entries will be lost.

New Source
Each input file from which cells are to be extracted, termed a "Source", has a separate page in the **Merge Tool** display. At startup, there is one empty source page, which is specified as "Source 1" in the tab at the top of the display. The **New Source** button will create a new empty source page, with a new tab with a unique name. The new page will become the visible page. Other source pages can be selected by clicking on the tabs. Each source page must be filled in with the appropriate entries before a merge can be performed.

Remove Source
Pressing this button will irretrievably delete the currently visible source page, if it is not the initial "Source 1" page. The page and its tab and contents will disappear.

New Toplevel
This will add the name of a top level cell to the **Top-Level Cells** list of the current page. These are cells that represent the top level of hierarchies to be extracted from the archive file named in the **Path to Source** entry on the same page. The names in the list must match an actual cell name found in the file. These are "top-level" in the extraction sense and need not be top-level in the overall cell hierarchy of the file. The list can contain the same name multiple times if multiple instances of the cell are needed in output.

Note that the user must have knowledge of the names of the cells used in the file. The names specified must be the actual names found in the file, and do not reflect name changes that might be applied during processing.

Remove Toplevel
This will remove the highlighted entry in the **Top-Level Cells** list, if an entry is highlighted. An entry is highlighted by clicking on it with the mouse. When an entry is removed, it will not appear in the output.

14.12.12 The Help Menu

The **Help** menu provides access to **Merge Tool** on-line documentation.

Help
This brings up the help system.

14.13 The Compare Layouts Button: Find Differences

The **Compare Layouts** button in the **Convert Menu** brings up the **Compare Layouts** panel. This is a graphical front-end for the **!compare** command, used to compare the contents of cells and hierarchies.

There are three different comparison modes, which can be selected with the notebook tabs at the top of the panel. The **Per-Cell Objects** mode will compare objects directly: box-to-box, poly-to-poly, etc. A difference will be recorded if an object does not have an identical counterpart in the other cell. In this mode only, there is provision for comparing the properties of the cells, objects and instances. In other modes, properties are ignored.

The **Per-Cell Geometry** mode will first convert the geometry to trapezoids, then compare the coverage of the trapezoid lists. Only differences in the actual dark-area will be reported. Both of these modes apply only to the geometry within a cell. The third mode, **Flat Geometry**, will compare the geometry after (logically) flattening the hierarchy. More detail will be provided below.

The lower half of the panel provides input areas for parameters that are used in any mode. The top two groups provide the sources to be compared. The **Source** entries can contain the name of a layout file in any of the supported formats, or the name of a Cell Hierarchy Digest (CHD) in memory. If left blank, the source is taken as the main database. Both **Source** entries may be blank in **Per-Cell Objects** mode, in order to compare cells in memory (in the current symbol table). The second **Source** entry can be left empty in any but the **Flat Geometry** mode, in which case the cells to compare must exist in memory, in the current symbol table. The top (left pointing) **Source** is the “reference” when the list of cells to compare is generated, so there is an asymmetry that should be kept in mind, which will be further discussed below.

If a file name is given as a source, a temporary CHD is created for use during the comparison, and is destroyed when the operation completes. Thus, when doing repeated comparisons, it is more efficient to create a CHD first, and reference this CHD for comparisons.

The actual list of cells to compare is generated from entries in the **Cells** and **Equiv** entry areas by logic to be described. These entry areas, if not blank, should contain space-separated cell names.

In many cases, there is only one list of cells to compare, and each cell is sought in both sources. If a cell is found in one source and not the other, this will appear in the log file, but is not considered to be an error. The cells list in this case is always given in the **Cells** entry.

If an **Equiv** list is given, there must be exactly the same number of entries given in the **Cells** list. The cells in the two lists will be compared term-by-term, in order. This is how one can compare cells with differing names. In all other cases, the **Equiv** list should be left blank. It is an error if **Equiv** entries are given with **Cells** blank, or if the list lengths differ. However, the **Equiv** list is ignored if in a per-cell comparison mode and **Recurse Into Hierarchy** is checked.

The interpretation of a blank **Cells** list depends on the comparison mode. If in flat comparison mode, or in a per-cell mode and the **Recurse Into Hierarchy** button is set, then the assumed cell list contains only the default cell from the top (left pointing) source. If this was a CHD name, the default cell is the one configured into the CHD, or the first top-level cell found in the source file. In the other cases, a blank **Cells** list is interpreted as all cells found in the top (left pointing) source.

In the special case that neither a left or right source is specified, then the **Cells** and **Equiv** lists can not be empty, and the names are cells in memory to compare.

In the per-cell modes with **Recurse Into Hierarchy** set, each entry in the **Cells** list is hierarchically expanded to a full list of the cells under the given cell, and these names are merged into a new list that contains no duplicates. If no **Cells** list was given, per the discussion above, the cell list is effectively the hierarchy of the default cell from the first source.

Below the source groups is a provision for layer-filtering. This is active when one of **Layers Only** or **Skip Layers** is pressed. The list contains space-separated layer names. With **Layers Only** active, only objects on the listed layers will be compared. With the **Skip Layers** button pressed (which deactivates **Layers Only** and vice-versa), only layers **not** listed will be considered. If neither button pressed, or if the layer list is empty, all layers will be considered.

During comparison, differences are recorded in an output file. By default, geometric differences are saved in a CIF-like format, providing lists of objects that appear in one cell but not the other. If the **Differ Only** check box is active, the geometric information is not written to the file, only the information that the cells differ.

The maximum number of differences that are recorded can be set with the **Maximum Differences** input area. If 0, then there is no limit. Otherwise, when the limit is reached, the comparison will terminate. It is usually advisable to set a limit, as an error in the source specification can potentially produce enormous output.

Pressing the **Go** button initiates comparison. When the job finishes, the user is given the option of viewing the log file. The log file is always named `diff.log` and is created in the current directory. An existing file of the same name is moved to a new name with a `.bak` extension added. The **!diffcells** command can be used to create cells from the log file for visualizing the differences.

The **Dismiss** button retires the panel. All entries are persistent, meaning that the panel will contain the same entered content the next time it appears.

14.13.1 Comparison Mode Pages

The comparison mode can be selected by clicking on the tabs at the top of the panel. Both of the per-cell modes contain **Recurse Into Hierarchy** and **Expand Array** buttons. The **Recurse Into Hierarchy** check box indicates that the cell to compare is to be taken as the top of a hierarchy, and this and all descendent cells should be compared. If not set, only the named cell is compared.

The **Expand Arrays** button applies when cell instances are being checked. When set, instance arrays are logically converted to individual placements before comparison. This avoids flagging differences that are due only to whether instances are arrayed or not, or whether that arraying is the same. This is useful when comparing OASIS files to GDSII files, for example, where the OASIS repetition finder may have been used.

Electrical cells can be compared using the **Per-Cell Objects** mode only. The mode to compare is selected on the page, which may be different from the current mode of the program.

When using **Per-Cell Objects**, one may select which type of objects to compare. Objects types that are not active are ignored. By default, text labels are ignored and all other objects are compared. A difference is indicated if a tested object does not have an identical counterpart in the other cell.

Comparison of labels can lead to false differences when comparing cells read from different file formats, since label bounding boxes are not well defined across file format conversion.

The **Per-Cell Objects** page contains a **Box to Wire/Poly Check** check box. With this mode selected, a two-vertex wire or four-vertex polygon that is rendered as a Manhattan rectangle will match a rectangle object with the same dimensions. Thus, files that have had these features converted to boxes to save space can be directly compared, without a lot of spurious entries in output.

The **Ignore Duplicates** check box in the **Per-Cell Objects** page sets a mode where if duplicate objects are present in one or both of the files, unmatched duplicates will not be reported if one of the duplicates has a match. Thus files with duplicates removed can be compared with the original file, and the duplicates will not appear in output as differences.

In **Per-Cell Geometry** mode, all boxes, polygons, and wires are included. Text labels are ignored. A button provides a choice whether or not to check subcells, which are tested as in the per-cell object mode.

When using **Per-Cell Geometry** mode, the geometry is compared within areas of a grid whose size is given by the PartitionSize variable. Experimenting with this size can lead to improved speed, depending on the layout density. The default partition size is 100 microns. For best performance, this can be increased for low density, or reduced for high density, where "density" refers to the number of trapezoids per area.

The **Flat Geometry** mode is somewhat orthogonal to the other modes. The algorithm uses two levels of gridding to partition the layout into pieces, and directly compares the geometry in each fine grid cell. This is very similar to the algorithm described for the `ChdIterateOverRegion` script function.

The fine grid size is entered in microns, the coarse grid size is entered as an integer multiple of the fine grid size. The flat geometry to render a coarse grid cell is held in memory, but subdivided into the fine grid cells for the comparison. Using a large coarse grid with a dense layout may trigger memory availability issues, yet using a large coarse grid usually improves speed. The user should experiment with the parameter values to see what works best with their layouts. The fine grid can be in the range of 1.0 to 100.0 microns, and the multiplier can be in the range $1 - 100$.

If the **Use Window** button is active, a rectangle entered into the entries (in microns) can be used to limit the comparison area. If not active, an area covering the entire bounding box of both cells being compared is used. The **S** and **R** buttons provide access to eight general purpose storage registers for rectangles, as provided in other panels that use rectangle data.

14.13.2 Property List Comparison

The **Per-Cell Objects** mode allows properties to be compared, unlike the other modes. There are three classes of properties: structure (cell) properties, cell instance properties, and object properties.

Whether or not to check properties can be set independently for each type of object. Properties of a given object type will only be compared when enabled by checking the boxes in the **Properties** group, plus the **Structure Properties** check box. When not checked, the properties of the corresponding object, cell instance, or the structure, will be ignored.

Property lists of objects and instances are only compared between otherwise identical objects or instances. Cell structure property lists will be compared whether or not other differences are found, when enabled.

There are three filters that can be applied, to reduce the number of properties compared. These correspond to cell properties, instance properties, and object properties. Further, different filtering is applied when comparing electrical and physical mode data. The **Property Filtering** option menu and **Setup** button control the filtering applied.

The **Default** choice of the menu applies default filtering. With this choice, there is no filtering (all properties considered) when comparing physical mode data. In electrical mode, the following defaults are applied:

Cell properties

Compare only PARAM, VIRTUAL, NEWMUT, SYMBLC, and NODMAP properties.

Instance properties
: Compare only MODEL, VALUE, PARAM, and NOPHYS properties.

Object properties
: Ignore all electrical properties of objects.

This filtering limits the comparison to properties over which the user has control, and whose differences are likely to indicate an actual design difference.

The **None** choice of the menu effectively turns filtering off, for both electrical and physical modes. This is comprehensive, but for electrical mode a lot of the internal properties, for example NODE properties, will be flagged as differing but may not represent a true difference in the design as the strings may include arbitrary internal assignments for some parameters.

The third possible menu choice, **Custom** allows the user to completely specify the filtering behavior. This is described in the next section. The filtering is specified from the pop-up produced by pressing the **Setup** button.

Properties are compared by number and string. In the output file, property comparison result lines are all in comment form (with '`#`' as the first character) so that they will be ignored if the file is subsequently processed with the **!diffcells** command. Property comparison results consist of a string indicating the cell, instance, or object containing the properties. If an instance or object, this is common to both input sources. Following this are listings of properties found in one source and not the other. Properties that are identical in the two sources are not listed.

14.13.3 Custom Property Filtering

The **Custom Property Filter Setup** panel is presented in response to pressing the **Setup** button in the **Per-Cell Objects** page of the **Compare Layouts** panel. The **Compare Layouts** panel is obtained from the **Compare Layouts** button in the **Convert Menu**.

This panel allows the user to set up the custom property filter strings for the cell, cell instances, and objects, for both electrical and physical mode comparisons. These filtering definitions are applied when layout comparison is being performed from the **Compare Layouts** panel in **Per-Cell Objects** mode, with the **Property Filtering** menu set to **Custom** and property checking enabled. The filtering also applies when using the **!compare** command, when neither of the `-f` or `-g` options is given, and the `-u` option is given and property checking is enabled.

The six entry areas correspond to six variables, which can (equivalently) be set directly. These variables are

PhysPrpFltCell
PhysPrpFltInst
PhysPrpFltObj
ElecPrpFltCell
ElecPrpFltInst
ElecPrpFltObj

If the entry area is empty, the corresponding variable is unset, and the default filtering will be applied. Otherwise, the string determines the filtering applied.

The strings consist of space and/or comma-separated lists of numbers or equivalent names. The names are simply mnemonics to the electrical properties, and are:

name	**value**
`model`	1
`value`	2
`param`	3
`other`	4
`nophys`	5
`virtual`	6
`bnode`	9
`node`	10
`name`	11
`labloc`	12
`mut`	13
`newmut`	14
`branch`	15
`labrf`	16
`mutlrf`	17
`symblc`	18
`nodmap`	19

Use of numbers and equivalent names is arbitrary and they can be mixed. Names will be recognized if at least the leading two characters are given, with enough additional characters so as to uniquely prefix one of the names in the list above. Names that are not recognized are silently ignored.

Specifying a list as described indicates that only the listed properties will be considered. However, it is possible to invert this logic.

If the first character in the string is '`s`', and the second character is not '`y`' (to avoid a clash with "`symblc`"), then the properties in the list that follows will be skipped, i.e., only properties not in the list will be considered. If the leading '`s`' is recognized as the "skip" indicator, all alphabetic characters up to the first delimiter or number will be stripped before parsing the list.

The recognition of names and the skip indicator are case-insensitive.

For example, the following specifications are all equivalent:

```
s1,2,3
skip1,2,3
skip,1,2,3
skip 1 2 3
skip,model,value,param
```

An empty entry area will trigger default filtering and is **not** an empty filter (blocking all). To provide an empty list, which blocks all properties from comparison, simply insert a character that is not recognized as a property number or '`s`'. Just about anything will do, one choice would be '`-`'. This will have the intended effect of setting up a filter with no elements, which will not match any values.

There is one more subtlety that may be encountered. In graphical mode, it is not possible to set the variables as booleans, i.e., to nothing. The graphical system will immediately unset the variable if this is attempted. However, in non-graphics mode, this won't happen, and the variables will take the null assignment. In this case, the corresponding filter will block all, rather than reverting to the default filter.

14.14 The Cut and Export Button: Export Cell Region

The **Cut and Export** button in the **Convert Menu** enables the user to define a rectangular area in a displayed layout, and export the flattened geometry in the area to a file. This can be useful for grabbing features of interest from the layout for documentation purposes or otherwise.

The user clicks twice or drags in a drawing window, to define a rectangle. The rectangle is automatically stored in register 0, of the eight rectangle registers that are available in pop-ups that use rectangle entry. Thus, pop-ups such as the **Format Conversion** panel, can load this rectangle by pressing the **R** button to the left of the window entry area, and selecting **Reg 0**.

After the rectangle is defined, the **Export Control** pop-up appears, preconfigured with the rectangle, and set for flattening, windowing, and clipping. The user may choose an output format and make any other desired changes, then press **Write File**. This will cause prompting for the name of the output file, which will be created if the user provides a valid name and no errors occur.

14.15 The Text Editor Button: Edit Cell Text

The **Text Editor** command brings up a text editor loaded with the text of the file for the current editing cell. This is only available for the ASCII text files: native and CIF. The text editor is described in 3.13.2.

This page intentionally left blank.

Chapter 15

The DRC Menu: Design Rule Checking

The **DRC Menu** contains commands which control checking of design rules. The menu is accessible only in physical mode, and design rule checking can only be applied in physical mode. *Xic* has the capability of checking for design rule violations as any object is created or modified, and for checking regions and cells interactively or in batch mode. The algorithm fully supports non-Manhattan geometry. Design rules are provided in the technology file, or interactively using the **Edit Rules** command.

The table below lists the commands found in the **DRC Menu**, and supplies the internal command name and a brief description.

DRC Menu			
Label	**Name**	**Pop-up**	**Function**
Setup	`limit`	**DRC Parameter Setup**	Set limits and other parameters
Set Skip Flags	`sflag`	none	Set skip flags
Enable Interactive	`intr`	none	Set interactive DRC
No Pop Up Errors	`nopop`	none	No interactive errors list
Batch Check	`check`	**DRC Run Control**	Initiate DRC run
Check In Region	`point`	none	Test rules in region
Clear Errors	`clear`	none	Erase error indicators
Query Errors	`query`	none	Print error messages
Dump Error File	`erdmp`	none	Dump errors to file
Update Highlighting	`erupd`	none	Update highlighting from file
Show Errors	`next`	sub-window	Sequentially display errors from file
Create Layer	`erlyr`	none	Write highlight error regions to objects on layer
Edit Rules	`dredt`	**Design Rule Editor**	Edit rules for layers

After a check is performed, violating objects are shown on-screen with the border highlighted, and a highlighting border is drawn around the test region containing the error. These objects are *not* removed from the database. It is up to the user to fix or ignore errors as they are indicated.

Presently, the indication of a violation is not saved as the cell is written.

Design rules are specified in the technology file, or with the **Design Rule Editor** made visible with the **Edit Rules** button in the **DRC Menu**. The rules are specified by a keyword, followed by an optional source region specification, followed by parameters. In addition to the built-in rule primitives to be described, a capability exists for users to define specialized or more complex tests.

15.1 Layer Expressions

Many of the design rules, extraction specifications, and commands make use of “layer expressions”. These expressions are used to signify regions of the layout where certain combinations of layers (or absence of layers) exist. A layer expression consists of a logical expression, in the format recognized by the script parser used to evaluate script files.

The expression may contain physical layer and derived layer (see 15.2) names, functions from the list below, operators from the table below, numeric constants, and parentheses to enforce precedence. In its simplest form, a layer expression is a layer name, which can be thought of as a list of regions corresponding to the dark areas (boxes, polygons, and wires) of that layer. A numeric value of zero represents emptiness, and a nonzero value represents full coverage.

When a layer expression is evaluated in the **!layer** command or the **Evaluate Layer Expression** panel, the result is always a normal layer, thus derived layers can be made visible by this means. If the layer expression represents a simple copy, the created physical layer will take any attributes of the derived layer (color, fill, etc.) that were given to the derived layer.

If the names of any defined layers are numeric values, one must be a little careful when specifying the equivalent numeric value, since a layer name interpretation will supersede a numeric interpretation. For example, in the presence of a layer named “1”, one could use “1.0” to specify the number 1. A four-digit hex number is always assumed to be a layer name, even if a layer of that name does not presently exist. This is necessary so that when reading the technology file, layer expressions can reference layers with numerical names (likely from GDSII conversion) that have not yet been defined. Layer names in the “decimal” format must be double quoted, e.g., `"22,0"`.

The layer name token can actually take an extended syntax which enables extraction of geometry from cells other than the current cell.

lname[`.`*stname*][`.`*cellname*]

See the description of the **!layer** command in 19.13.2 for a description of this syntax and examples.

The following operators are accepted in layer expressions:

& or ∗	intersection
| or +	union
!	inversion
ˆ	exclusive-or
−	and-not, i.e., $A - B = A\&!B$
`and`	synonym for &
`or`	synonym for |
`not`	synonym for ! or −
`xor`	synonym for ˆ

The operator-equivalent keywords (`and`, `or`, `not`, `xor`) are recognized without case sensitivity. The `not` keyword can represent a unary negation or a binary “andnot”, depending on the context. Thus, for layers `A` and `B`, each of the following are equivalent: `A not B`, `A − B`, `A &! B`, `A and not B`.

Parentheses can be used to enforce precedence.

The expression returns an internal data structure representing those regions of the current cell where the expression is true, i.e., where the layers exist with the given logic.

There is a special layer named "$$" which logically consists of boxes covering each of the subcells in the current cell.

The **!layer** command can create a new layer from a layer expression, and is therefor a good vehicle for experimenting with layer expressions.

The tokens are interpreted as they would be in an ordinary expression involving numbers, thus their precedence might not be quite as expected in layer expressions. For example

```
!layer CAA = !CAA & $$
```

and

```
!layer CAA = !CAA * $$
```

are *not* equivalent. The latter expression is equivalent to

```
!layer CAA = !(CAA & $$)
```

since '*' has higher precedence than '&'. The equivalent expression is

```
!layer CAA = (!CAA) * $$
```

(recall that '$$' is the name for an internal layer consisting of subcell bounding boxes).

The following function calls are supported in layer expressions. Only the functions listed below are available, and all return a layer expression object.

`sqz`(layer_exp *expr*)
: This is a special function that evaluates the layer expression passed as an argument, but the geometry for the given layers is obtained from the selection queue (the currently selected objects), and not the entire cell as in the normal case. It can be freely used within a larger layer expression.

  Below are some examples, using the **!layer** command.

  `!layer new = sqz(CPG-CAA)`
  : Create a layer "new" that will contain the selected objects on CPG clipped around selected objects on CAA.

  `!layer new = VIA & sqz(M2)`
  : Create a layer "new" that will contain the areas of VIA that overlap selected objects on M2.

  `!layer CPG = CPG - sqz(temp)`
  : Clip out the selected objects on layer temp from CPG.

`bloat`(real *incr*, layer_exp *layer*, int *mode*)
: This expands the features on the layer by *incr* (in microns), which may be negative. The effect is similar to the **!bloat** command and the `BloatZ` script function. The *mode* integer is described with the **!bloat** command.

`extent`(layer_exp *layer*)
: This evaluates to a trapezoid list containing at most one entry, a rectangle giving the bounding box of the expression result. The return is null if the expression is nowhere dark. This is similar to the `ExtentZ` script function.

`edges`(real *incr*, layer_exp *layer*, int *mode*)
: This creates an edge list, similar to the `EdgesZ` script function. See the description of that function for the edge modes available. The modes 0–3 are equivalent to returns from the `bloat` function when returning the edge template, for the four corner fill-in modes.

`manhattanize`(real *dimen*, layer_exp *layer*, int *mode*)
: This converts the representation to a Manhattan approximation. The first argument is the minimum width or height in microns of rectangles that are created to approximate the non-Manhattan parts. The third argument is an integer taken as zero or nonzero to specify which of two algorithms to use. This is similar to the **!manh** command (where the algorithms are described), and to the `ManhattanizeZ` script function.

`box`(real *l*, real *b*, real *r*, real *t*)
: This defines a rectangular region from the four real arguments, which can be used for clipping or construction in layer expressions. The coordinates are given in microns. This is similar to the `BoxZ` script function.

`zoid`(real *xll*, real *xlr*, real *yl*, real *xul*, real *xur*, real *yu*)
: This defines a horizontal trapezoid region from the six real arguments, which can be used for clipping or construction in layer expressions. The coordinates are given in microns. This is similar to the `ZoidZ` script function.

`filt`(layer_exp *zoids*, layer_exp *lyr2*)
: This function is rather specialized. First, the trapezoids passed in the first argument are separated into groups of mutually-connected trapezoids. Each group is like a wire net. We throw out the groups that do not intersect with nonzero area the dark area implied by the second argument. The return value is a list of the trapezoids that remain.

`geomAnd`(layer_exp *lyr1* [, layer_exp *lyr2*])
: If one argument is given, the result is the overlapping parts of regions in the internal list corresponding to the argument. This is only useful if the argument was explicitly constructed with `geomCat` (see below). With two arguments, this is equivalent to the intersection operator. The function is similar to the `GeomAnd` script function.

`geomAndNot`(layer_exp *lyr1*, layer_exp *lyr2*)
: This is equivalent to the and-not operator, and is similar to the `GeomAndNot` script function.

`geomCat`(layer_exp *lyr1*, ...)
: This takes one or more layer expression arguments and simply concatenates the regions, without any merging or clipping, similar to the `GeomCat` script function.

`geomNot`(layer_exp *lyr*)
: This is equivalent to the inversion operator, similar to the `GeomNot` script function.

`geomOr`(layer_exp *lyr1*, ...)
: This takes one or more layer expression arguments and returns the union, constructed internally so that no two regions overlap. This is similar to the `GeomOr` script function.

`geomXor`(layer_exp *lyr1* [, layer_exp *lyr2*])
: If one argument is given, the return is the set of regions representing the exclusive-or of regions

represented by the argument. This is only useful if the user has explicitly constructed the argument using `geomCat`. If two arguments are given, the result is the exclusive-or of the areas, equivalent to the exclusive-or operator. This function is similar to the `GeomXor` script function.

`drcZlist`(string *layername*, string *rulename*, integer *index*, layer_exp *lyr1*)
This will return the test areas based on an existing design rule definition, very similar to the `DRCzList` script function. This function exists only when design rule checking is included in the feature set.

`drcZlistEx`(layer_exp *lyr1*, tring *target*, string *inside*, string *outside*, integer *incode*, integer *outcode*, real *dimen*)
This will return the test areas based on the DRC test area generation specified by the arguments, very similar to the `DRCzListEx` script function. This function exists only when design rule checking is included in the feature set.

Examples:

```
!layer M2 = M2 & box(100, 100, 200, 200)
```

This clips M2 to the given box.

```
!layer M2 = bloat(5, M2, 0)
```

This bloats the M2 geometry by 5 microns.

15.2 Derived Layers

Derived layers are layers which represent the result of a layer expression involving normal layers and other derived layers. Although derived layers are invisible, they can be used to create normal layers which can be seen. Derived layers were developed forthe design rule checking (DRC) system, but can be used in any layer expression.

There are actually two implementations of derived layer functionality. In the original implementation, developed for the DRC system, the geometry of derived layers must be created or updated before the derived layer is referenced. In use, reference to a derived layer in a layer expression retrieves this geometry, very similar to what happens when a normal layer is referenced. Ordinarily, the derived layer geometry will be cleared after final use. The DRC system handles creation and destruction of derived layer geometry transparently.

In the second mode of operation, when the parse tree for the derived layer is created, references to derived layers will be recursively parsed and stitched into the tree. The final parse tree will contain normal layers only, and can therefor be evaluated in any context, without the need for precomputed geometry caches. This method is more convenient and flexible, however the original method may run more quickly, particularly when there are a large number of evaluateions of the same expression, as in DRC.

The derived layer evaluation mode is invisible to the used except tehat it can be specifically set when using the derived layer script library function interface. This provides explicit control of derived layer geometry creation and destruction, and evaluation mode, through functions like `EvalDerivedLayers` and `ClearDerivedLayers`.

Derived layers can be defined in the technology file, or the definitions may be imported from a Virtuoso ASCII technology file. Derived layers can also be created with the `AddDerivedLayer` script function.

A derived layer definition consists of a layer name, a positive integer index number, and a layer expression string. The layer expression can not be null or empty. The index number is used for establishing the order when the derived layers are listed, such as when printing an updated technology file. This number need not be unique among derived layers, derived layers with the same index are ordered alphabetically by name.

The derived layer name may be in the *layer*:*purpose* form, or may be a simple alphanumeric name. It either case, it is treated as a case-insensitive atomic token in name comparisons. This is subtly different from normal layers, where for example "`M1`" and "`M1:drawing`" refer to the same *Xic* layer (the `drawing` purpose is the default and need not be explicitly specified). As derived layer names, the two forms represent two different derived layers.

Derived layers can be created arbitrarily. If a name is already in use, the existing derived layer definition is updated. It is not an error if a derived layer has a name that matches a normal layer. References to that name will resolve to the normal layer, thus with a few exceptions the derived layer would be inaccessible.

15.3 Built-In Design Rules

Xic provides a number of internal rule evaluation functions, to be described in this section. These should cover basic and common design rules as published for a particular fabrication process. More complex rules can perhaps be accommodated with the user-defined rule capability.

Design rules are associated with *Xic* physical and derived layers. In the technology file, the rule definitions appear in layer blocks for physical and derived layers.

The rules, and derived layers, make use of layer expressions. A layer expression can be a single layer name, or a more complicated expression involving other normal and derived layer names. In a rule specification, the expression syntactically represents a single token, though the expression may include white space. The expression in the specification is parsed as far as possible (white space is ignored), and the rest of the line is taken as further input to the specification.

The result of the evaluation of a layer expression can be thought of as a set of geometric figures representing areas where the expression is true. Below are two example rule specifications that use layer expressions.

```
Overlap M1 | M2 #layer must be covered by M1 or M2
NoOverlap Via&!M1 #layer must never overlap Via without M1
```

Where a layer expression can be used, a derived layer can also be used. The examples above can be expressed alternatively using derived layers.

```
DerivedLayer m1orm2 M1 | M2
DerivedLayer vianotm1 Via&!M1
...
Overlap m1orm2 #layer must be covered by M1 or M2
NoOverlap vianotm1 #layer can't overlap Via without M1
```

Whether it is "better" to use layer expressions or derived layers in the rules is still a bit open, as derived layers are a new feature. There may be performance differences, as evaluation is quite different. In the case of derived layers, all geometry on the derived layers is computed before a DRC run, and cleared after the run. Thus, during rule evaluation, existing geometry is simply accessed. When a layer expression is used, the expression is evaluated in test regions while the rule is being evaluated. Thus, the expression requires evauation, over a tiny area, many times. It is not clear that one method or the other would be generally faster, users should experiment. Use of layer expressions may be preferred if memory is constraining, as the amount of memory required to save derived layer geometry may be substantial.

Use of derived layers may be required for certain types of rules. For example, suppose that we have a constraint:

```
(NP or PP) Enclosure of PO 0.15
```

What this means is that layers NP or PP must cover layer PO, with 0.15 microns distance surrounding PO covered by NP or PP. This translates directly to the `MinNoOverlap` rule, but applied on the layer combination NP|PP, which can be accomplished with a derived layer.

```
DerivedLayer implant NP|PP
MinNoOverlap PO 0.15 # (NP or PP) Enclosure of PO 0.15
```

Ordinarily, a design rule evaluation proceeds as follows. All evaluation is performed using a "pseudo-flat" representation of the cell hierarchy, which effectively translates the coordinates of every object in the hierarchy to the space of the top-level cell. Each object in this space can be tested without having to know which cell in the hierarchy actually contains the object. The "global" tests, that are not associated with individual objects, such as checking for holes, are done first. Then, the per-object tests are performed on each object in the pseudo-flat representation. For each object (box, polygon, or wire), each test listed for the layer of the object is run in sequence. The per-area tests, which are done first, are applied to the area of the object, and remaining tests are applied to constructed regions along each edge of the object.

Below are descriptions of the built-in design rule test functions, and the syntax used to specify the test in a layer block in the technology file. Each rule line starts with the defining keyword, followed by an optional Region expression, required parameters, and an optional explanation string.

If the Region keyword and associated expression are given in the rule specification, the source area becomes those regions where the expression is true, within the boundaries of the object. The per-area tests are applied to the areas where the expression is true, and the other tests are applied to the edges of these regions. In simple cases, the Region expression is not necessary, but it does provide additional capability for more complex testing.

Use of `Region` is very similar to defining the rule on a derived layer consisting of the original layer ANDed with the `Region` expression.

An optional descriptive string can follow the rule specification. This string will be saved and included in violation reports. It is a good idea start the explanation string (if any) with the script comment character '`#`' to guarantee termination of the preceding expression. Recall that white space is ignored when parsing the expression. Most of the time, the parser can recognize the end of the expression, so the comment character is not necessary, but it is possible that the explanation string might start with an operator token such as '`*`' or a reserved keyword such as "`not`", and the expression parse would fail.

For certain rules, the description may have multiple components, i.e., it actually consists of multiple strings. This syntax will be described below for the affected rules, but is amounts to simply double-quoting the individual strings. When constraints are imported from a Virtuoso ASCII technology file, there are occasions where multiple constraints, each with a description string, map to a single *Xic* primitive rule. These strings will be recovered when converting back to Virtuoso format with the **!dumpcds** command.

In the discussion that follows, the following definitions will be used. An "object" is a physical entity found in the database. A "figure" is a geometrical shape and an associated layer expression which is true within the shape. A figure can represent an object and the object's layer, for example, or one of the regions where a layer expression is true, and the layer expression. The "source" is a set of figures where rule evaluation is to be performed. If no Region is given, the source is simply the figure representing the object's geometry and the object's layer. Otherwise, the source is the set of figures where the region expression is true within the object. Two or more figures are "compatible" if they are associated with the same layer expression.

15.3.1 Global Rules

The first two rules operate differently from the others, in that they do not operate on a per-object basis, rather they operate on an entire pseudo-flattened layer. As such, they can be computationally and memory intensive. These "global" tests are performed before the others, however they are performed only if the area being checked is the entire cell area.

Connected **Rule**

Syntax: `Connected` [`Region` *region_expr*] [*string*]

If given in the layer block, the layer or region description (which is applied to the whole layer) is tested to see that all figures are mutually connected (touch or overlap). Disjoint groups of figures are flagged as violations in the top level cell. The group with the largest area is assumed to be the "correct" group.

NoHoles **Rule**

Syntax: `NoHoles` [`Region` *region_expr*] [`MinArea` *area*] [`MinWidth` *width*] [*string*]

If given in the layer block, the layer or region description (which is applied to the whole layer) is tested for clear area surrounded by dark area. Each such area is optionally tested. If the `MinArea` is given and positive and the clear area is smaller, a violation will be reported. If the `MinWidth` is given and positive, the clear area must be large enough so that for any edge, a rectangular projection along the edge extending into the interior by the given width will be clear. If not, a violation will be reported. If neither of the `MinArea` or `MinWidth` are given, then any such clear area found will be flagged as a violation.

The `MinArea` and `MinWidth` clauses are set by the `minHoleArea` and `minHoleWidth` constraints when importing Virtuoso technology data. Each constraint may have a separate reference string. To keep these distinguishable, the *string* can actually be three double quoted strings, e.g., the form is

```
"# rule description" "minHoleArea string" "minHoleWidth string"
```

This guarantees that the original reference strings are regenerated when the **!dumpcds** command is used to generate a Virtuoso technology file. If a component string doesn't exist, one can use `""` (two double-quote marks) as a placeholder. Strings to the right that don't exist can be skipped entirely.

15.3.2 Area Rules

The following are the per-area tests, and are applied to the area of each source figure, for each object in the pseudo-flat representation.

Exist Rule

Syntax: `Exist` [*string*]

This rule will indicate a violation if any dark area is found on the layer containing the rule. Unlike most if not all other rules, no Region specification is allowed.

The Exist rule is intended for derived layers whose construction would indicate an incorrect combination of other layers (normal and derived). Layer expressions and derived layers can be used as alternatives to many of the built in rules, and for formulating new rules. The results are a bit different from the per-object and per-edge iteration of the normal rule evaluation flow. All violations are found as objects on the derived layer, there is no search limit (e.g., the normal flow may limit reporting to one violation per object, though an object may be associated with multiple violations). The approach gives the rule-author flexibility.

Overlap Rule

Syntax: `Overlap` [`Region` *region_expr*] *expression* [*string*]

This test fails if any source figure is not completely covered by the figures associated with the *expression*. In other words, for the situation where no Region is given, the *expression* must evaluate true at every point of every object on the present layer. This is illustrated in Figure 15.1, for no Region and an expression consisting of a single layer.

IfOverlap Rule

Syntax: `IfOverlap` [`Region` *region_expr*] *expression* [*string*]

This test fails if any source figure is partially covered by the figures associated with the *expression*. Unlike the Overlap keyword, this test does not fail if there is no intersection. The *expression* must be either always true or always false at every point of a source figure, or for every object on the present layer if no Region is given. Figure 15.2 illustrates use of this keyword, for no Region and an expression consisting of a single layer.

NoOverlap Rule

Figure 15.1: The Overlap test. The present figure (solid) must be completely covered by figures resulting from evaluating the expression argument (dotted).

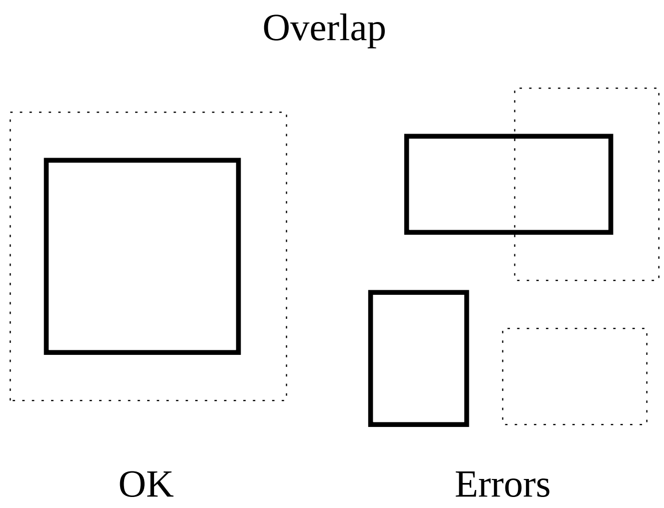


Figure 15.2: The IfOverlap test. The present figure (solid) can not be partially covered by figures resulting from evaluating the expression argument (dotted).

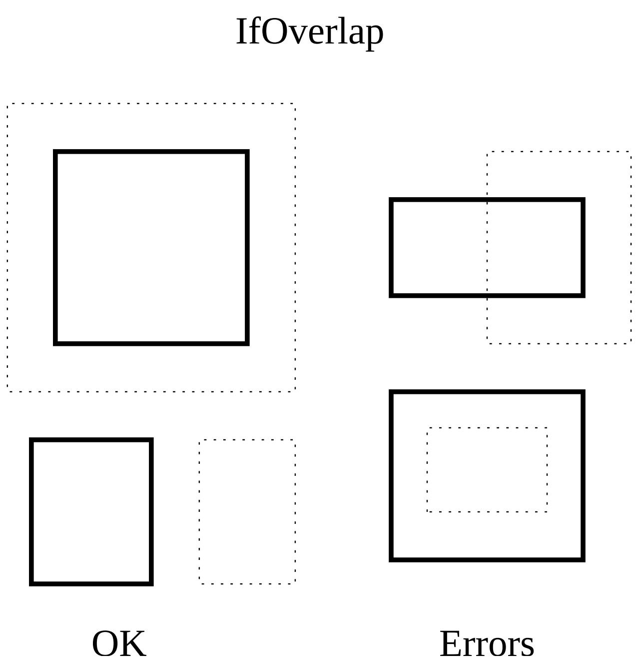
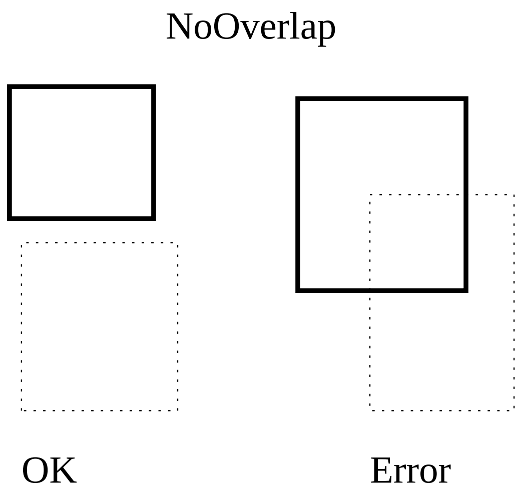


Figure 15.3: The NoOverlap test. The present figure (solid) can not intersect with figures resulting from evaluating the expression argument (dotted).



Syntax: `NoOverlap` [`Region` *region_expr*] *expression* [*string*]

This test fails if any source figure has non-zero intersection area with the figures associated the *expression*. The *expression* must evaluate false at every point of every source figure. This is illustrated in Figure 15.3, for no Region and an expression consisting of a single layer.

AnyOverlap **Rule**

Syntax: `AnyOverlap` [`Region` *region_expr*] *expression* [*string*]

The AnyOverlap test signals a violation if any source figure has no intersection area with the figures associated with the *expression*. This is illustrated in Figure 15.4, for no Region and an expression consisting of a single layer.

PartOverlap **Rule**

Syntax: `PartOverlap` [`Region` *region_expr*] *expression* [*string*]

The PartOverlap test signals a violation if any source figure is either completely covered or completely uncovered by the figures associated with the *expression*. This is illustrated in Figure 15.5, for no Region and an expression consisting of a single layer.

AnyNoOverlap **Rule**

Figure 15.4: The AnyOverlap test. The present figure (solid) must be partially or fully covered by figures resulting from evaluating the expression argument (dotted).

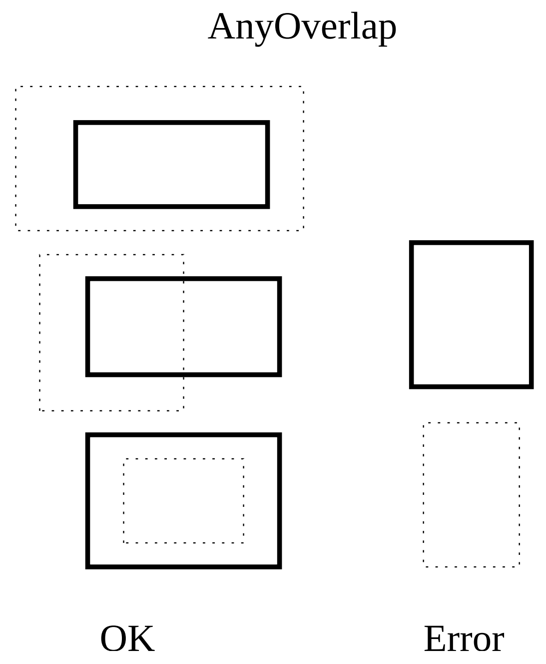


Figure 15.5: The PartOverlap test. The present figure (solid) must be partially covered by figures resulting from evaluating the expression argument (dotted).

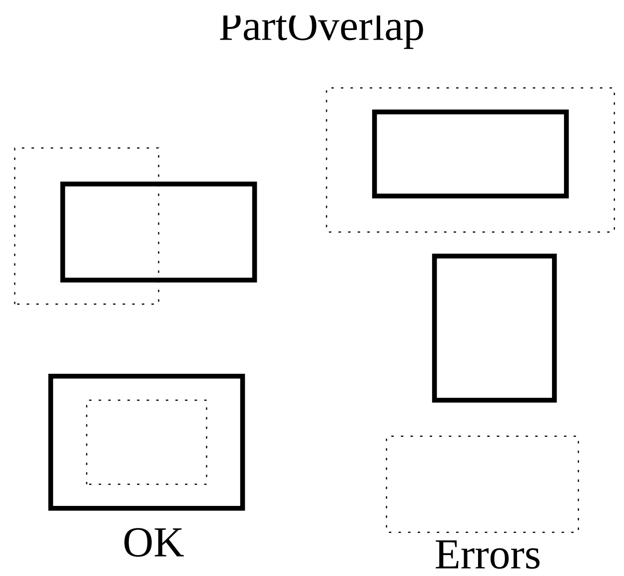
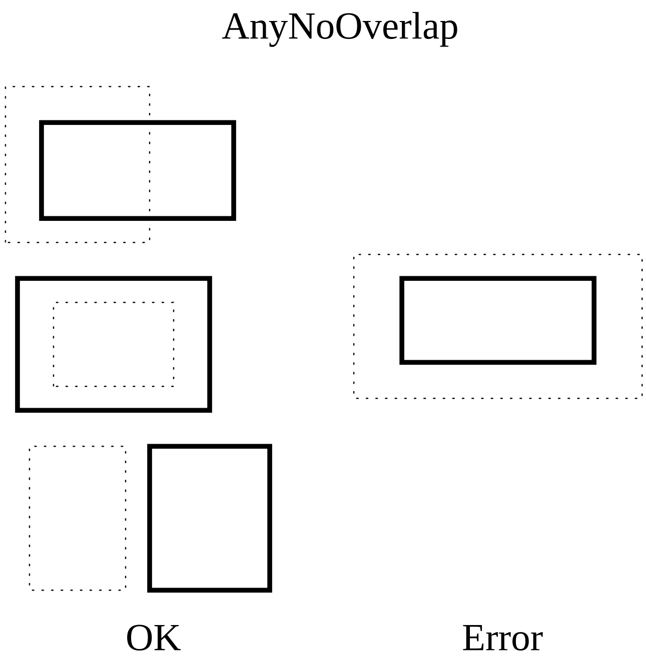


Figure 15.6: The AnyNoOverlap test. The present figure (solid) must be partially uncovered by figures resulting from evaluating the expression argument (dotted).



Syntax: `AnyNoOverlap` [`Region` *region_expr*] *expression* [*string*]

The AnyNoOverlap test signals a violation if any source figure is completely covered by the figures associated with the *expression*. This is illustrated in Figure 15.6, for no Region and an expression consisting of a single layer.

The returns from the various Overlap tests are summarized in the table below.

rule	total coverage	partial coverage	no coverage
Overlap	ok	error	error
IfOverlap	ok	error	ok
NoOverlap	error	error	ok
AnyOverlap	ok	ok	error
PartOverlap	error	ok	error
AnyNoOverlap	error	ok	ok

MinArea **Rule**

Syntax: `MinArea` [`Region` *region_expr*] *area* [*string*]

For each object tested, the neighborhood of the object is searched for mutually touching, source compatible objects. The area covered by the objects is computed, and this is compared with the given area (which is given in square microns). If the computed area is less than the test value a DRC violation is indicated.

When importing Virtuoso technology data, the `minArea` constraint maps directly to this rule.

MaxArea **Rule**

Syntax: `MaxArea` [`Region` *region_expr*] *area* [*string*]

The total area of the source figures is compared with the given area (which is given in square microns). If the area of the figures is greater than the test value a DRC violation is indicated. The area is measured on a per-object basis, and is the sum if there are multiple figures (due to a region expression). This does not account for adjacent objects.

15.3.3 Edge Rules

In the discussion to follow, the "source" is the material of the object being checked, either a layer (on which the rule is defined) or a layer expression result if the `Region` specification is given. The "target" is the set of figures associated with the *expression* supplied to the rule, for those rules that take an *expression*.

The rules described in this section are "edge tests" where the region of interest is generally a small constructed area along an edge. The test is applied for each applicable edge portion of each source figure. The constructed area is rectangular, parallel to the source figure edge, and may extend out of the source figure, or into the source figure. These extend only along the parts of the source figure edge where certain conditions apply, as will be described. The width of the test area (perpendicular to the figure edge) is the dimension associated with the rule.

We find the edge portions of a source figure as follows. We iterate through the edges. For each edge, we construct a unit-width test area that extends outside of the figure. We throw out the parts of the edge where the test area intersects "source compatible figures", meaning the same layer or layer expresion as the source. Thus we throw out the part of the edge which is not really an edge, but a boundary between dark areas of the same type and is therefor not a physical discontinuity. The resulting edge portion is the starting point for further restrictions that are rule-specific.

Below is a table which identifies the edge portions identified for the built-in rules, and the test performed.

rule	**where**	**test**	**src out**	**trg in**	**trg out**
`MinEdgeLength`	in	len	u	c	c
`MaxWidth`	in	snf	u		
`MinWidth`	in	sf	u		
`MinSpace`	out	se	u		
`MinSpaceTo`	out	te	u	u	x
`MinSpaceFrom`	out	tf	u	c	x
`MinOverlap`	in	tf	u	c	x
`MinNoOverlap`	in	te	u	u	x

The first column specifies the built-in rule name of the "edge" rules. These will be described in more detail below. The **where** column indicates the direction of the constructed test area along a source figure edge, either projecting into the figure or outside of the figure. The **test** indicates the type of test to perform in the constructed area. These are

len
: Measure the edge length and compare to given dimension.

snf
: The test area is not fully covered by source-compatible material.

sf
: The test area is fully covered by source-compatible material.

se
: The test area contains no source-compatible material.

tf
: The test area is fully covered by target-compatible material.

te
: The test area contains no target-compatible material.

The three remaining columns indicate the part of the source figure edge that is used to construct the test area. These indicate the required coverage of the source material just outside the edge, and target material just inside and just outside of the edge. The possibilities are

c
: Covered by the material.

u
: Not covered by the material.

x
: Doesn't matter.

So, for example, for `MinSpaceTo`, we take the part of the edges that are not covered by source just outside of the edge, and not covered by target just inside of the edge. The test will pass if the constructed area, which extends out of the figure, intersects no target material.

Each of the "edge" rules recognize two additional keywords:

`Outside` *layer_expr*
: This will apply when identifying the part of the edge of a source figure to use when constructing test areas. The given layer expression must be dark along the edge just outside of the figure, in the parts of the edge to use for the test area. This provides an additional rather arbitrary constraint on the test area construction which may be of use in some cases.

`Inside` *layer_expr*
: This is very similar to the `Outside` constraint, but applies to the side of the edge just inside of the figure. Only the parts of the edge where the given layer expression is dark just inside of the figure are considered for edges of the test area.

For example:

> "The minimum distance to CO from a NP/PP butt edge over OD is 0.06 microns."

This can be implemented as

```
PhysLayer NP
MinSpaceTo Region OD Inside !PP Outside PP CO 0.06
PhysLayer PP
MinSpaceTo Region OD Inside !NP Outside NP CO 0.06
```

The built-in edge rules are described in more detail below. In each, for simplicity we will use the following as an abbreviation for the syntax elements:

EdgeArgs = [`Region` *region_expr*] [`Inside` *inside_expr*] [`Outside` *outside_expr*]

MinEdgeLength Rule

Syntax: `MinEdgeLength` [*EdgeArgs*] *expression length* [*string*]

This test checks the length of the edges where source and target figures intersect. For each edge of the source figure, the parts of the edge where *expression* is true on both sides of the edge are considered. If the length of the part is less than the given *length*, a violation is flagged.

Example:

Rule: "M3 width must be 2 microns or greater when crossing over M2 edges."

This can be handled in two ways. The first method is to put the rule in the M2 block:

```
Layer M2
...
MinEdgeLength M3 2
```

The second approach is to put a slightly different implementation into the M3 block. This has a problem in that if the M3 is composed of several objects which together provide the minimum edge length, this test will fail since it looks at the objects individually.

```
Layer M3
...
MinEdgeLength Region M2 M3 2
```

MaxWidth Rule

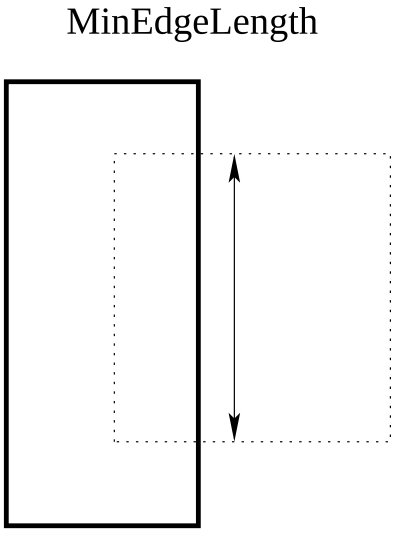
Syntax: `MaxWidth` [*EdgeArgs*] *width_in_microns* [*string*]

For the parts of each edge of the source that are not coincident or overlapping with source-compatible figures, and the edge length is greater than the given dimension, a rectangle extending normally from the edge into the source figure by the given dimension plus a tiny extra is constructed. The test fails if this constructed rectangle is completely covered by source-compatible figures.

The "tiny extra" is one internal unit for Manhattan edges. An additional "fudge factor" is added for non-Manhattan edges to overcome roundoff error.

When importing Virtuoso technology data, the `maxWidth` constraint maps directly to this rule.

Figure 15.7: The MinEdgeLength test. The length of the intersecting edge of the present figure (solid) and the target (dotted) must be greater than the value given.



MinWidth **Rule**

Syntax: `MinWidth` [*EdgeArgs*] *width_in_microns* [`Diagonal` *alt_width*] [*string*]

For the parts of each edge of the source that are not coincident or overlapping with source-compatible figures, a rectangle extending normally from the edge into the source figure by the given dimension is constructed. The test fails if this constructed rectangle is not completely covered by source-compatible figures. Note that the angle formed by two adjacent edges of a figure measured inside of the figure must be 90 degrees or larger, i.e., this rule prevents acute angles in polygons. Figure 15.8 illustrates the test performed under this keyword, for no Region.

If the `Diagonal` clause is given and the *alt_width* is positive, the *alt_width* will be used when the edge being tested is nonorthogonal.

The MinWidth test also fails if the length of a line defined by the overlap points of two mutually overlapping corners of a source figure and another compatible figure is less than the given dimension, including the condition where corners of the two figures touch but the intersection area is zero.

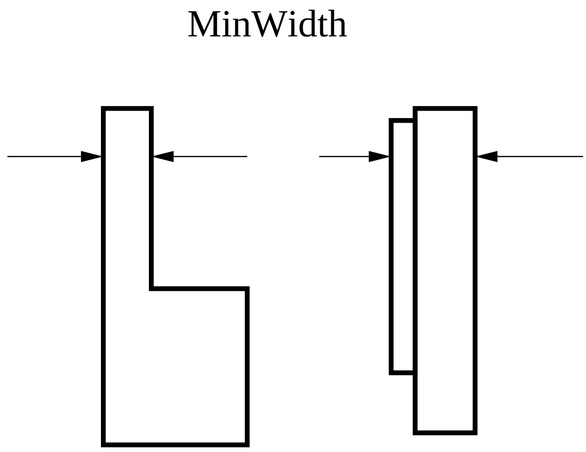
When importing Virtuoso technology data, the `minWidth` constraint maps directly to this rule.

The `Diagonal` clause is set by `minDiagonalWidth` constraint when importing virtuoso technology data. This may have its own reference string. To keep this distinguishable, the *string* can actually be two double-quoted strings, e.g., the form is

```
"# rule description" "minDiagonalWidth string"
```

This guarantees that the original reference strings are regenerated when the **!dumpcds** command is used to generate a Virtuoso technology file. If a component string doesn't exist, one can use `""` (two double-quote marks) as a placeholder. Strings to the right that don't exist can be skipped entirely.

Figure 15.8: The MinWidth test. The edge-to-edge spacing across a region on the present layer must not be less than the given dimension.



MinSpace **Rule**

Syntax: `MinSpace` [*EdgeArgs*] *space_in_microns* | `SpacingTable` *table definition* [`Diagonal` *diag_space*] [`SameNet` *snet_space*] [*string*]

For the parts of each edge of the source that are not coincident or overlapping with source-compatible figures, a rectangle extending normally from the edge out of the source figure by the given dimension is constructed. The test fails if the constructed rectangle has nonzero intersection area with source-compatible figures. Note that the angle formed by two adjacent edges of a figure measured outside the figure must be 90 degrees or greater, i.e., this rule prevents acute notches in polygons, and acute bends in wires. Figure 15.9 illustrates the test performed under this keyword, for no Region.

If a spacing table (see 15.4) is specified and active, the dimension used is computed from the spacing table, for Manhattan edges. An inactive table will still supply the default spacing, which is taken as the *space_in_microns*.

If the `Diagonal` clause is given with positive *diag_space*, then the *diag_space* value will be used when the edge being tested is nonorthogonal.

The `SameNet` clause is currently not implemented, and its presence has no effect.

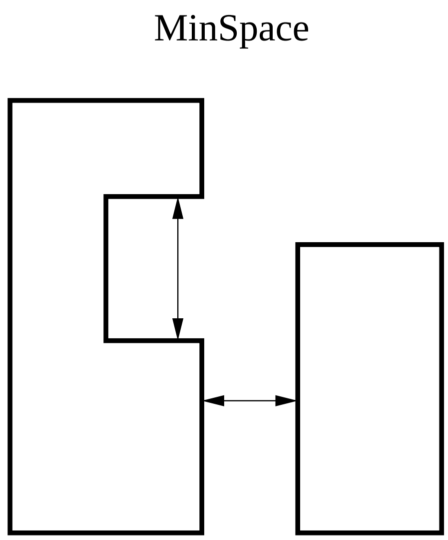
The MinSpace test also fails if the space from a corner of a source figure to another non-touching compatible figure is less than the dimension.

When importing Virtuoso technology data, the `minSpacing` single-layer constraint maps directly to this rule.

The `Diagonal` and `SameNet` clauses are set by `minDiagonalSpacing` and `minSameNetSpacing` single-layer constraints when importing virtuoso technology data. These may have their own reference strings. To keep these distinguishable, the *string* can actually be three double-quoted strings, e.g., the form is

```
"# rule description" "minDiagonalSpacing string" "minSameNetSpacing string"
```

Figure 15.9: The MinSpace test. The edge-to-edge spacing between regions on the present layer must not be less than the given dimension.



This guarantees that the original reference strings are regenerated when the **!dumpcds** command is used to generate a Virtuoso technology file. If a component string doesn't exist, one can use `""` (two double-quote marks) as a placeholder. Strings to the right that don't exist can be skipped entirely.

MinSpaceTo **Rule**

Syntax: `MinSpaceTo` [*EdgeArgs*] *expression space_in_microns* | `SpacingTable` *table definition* [`Diagonal` *diag_space*] [`SameNet` *snet_space*] [*string*]

For the parts of each edge of the source that are not coincident or overlapping with source-compatible figures or with target figures which extend into the interior of the source figure, a rectangle extending normally from the edge out of the source figure by the given dimension is constructed. The test fails if the constructed rectangle has nonzero intersection area with target figures. Note that overlap of the two figures is never flagged as a MinSpaceTo violation, but touching figures will generate a violation. Figure 15.10 illustrates the test performed under this keyword, for no Region and an *expression* consisting of a single layer.

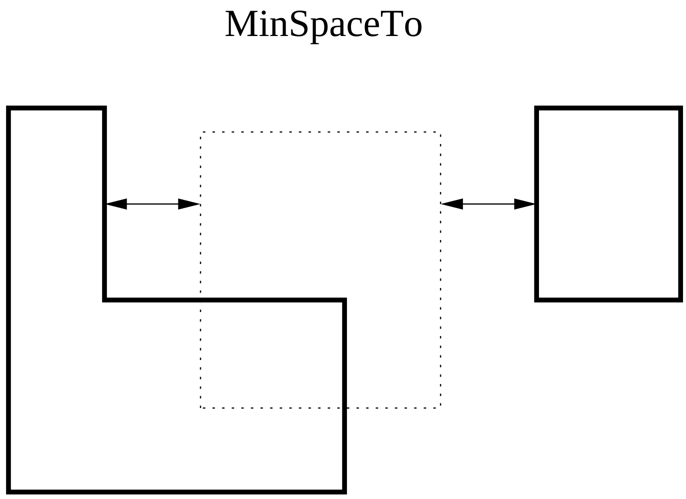
If a spacing table (see 15.4) is specified and active, the dimension used is computed from the spacing table, for Manhattan edges. An inactive table will still supply the default spacing, which is taken as the *space_in_microns*.

If the `Diagonal` clause is given with positive *diag_space*, then the *diag_space* value will be used when the edge being tested is nonorthogonal.

The `SameNet` clause is currently not implemented, and its presence has no effect.

The MinSpaceTo test also fails if the distance from a corner of the source figure to a non-touching target figure is less than the dimension. The corner test is skipped if the corner point is on the edge of or internal to another figure compatible with either the source or the *expression*.

Figure 15.10: The MinSpaceTo test. The minimum edge-to-edge spacing between regions of the present layer (solid) and the argument layer (dotted) must not be less than the given dimension.



When importing Virtuoso technology data, the `minSpacing` two-layer constraint maps directly to this rule.

The `Diagonal` and `SameNet` clauses are set by `minDiagonalSpacing` and `minSameNetSpacing` two-layer constraints when importing virtuoso technology data. These may have their own reference strings. To keep these distinguishable, the *string* can actually be three double-quoted strings, e.g., the form is

```
"# rule description" "minDiagonalSpacing string" "minSameNetSpacing string"
```

This guarantees that the original reference strings are regenerated when the **!dumpcds** command is used to generate a Virtuoso technology file. If a component string doesn't exist, one can use `""` (two double-quote marks) as a placeholder. Strings to the right that don't exist can be skipped entirely.

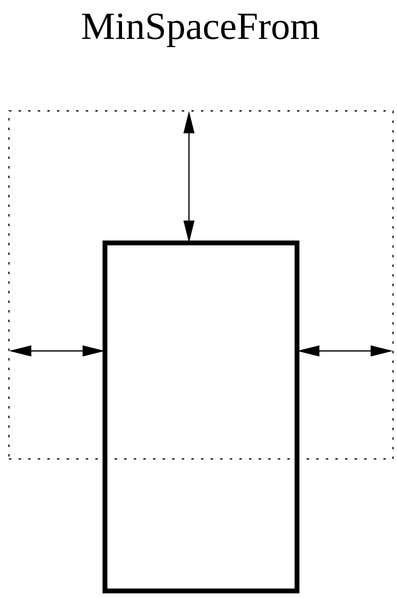
MinSpaceFrom **Rule**

Syntax: `MinSpaceFrom` [*EdgeArgs*] *expression dimension_in_microns* [`Enclosed` *enc_dimen*] [`Opposite` *dimen1 dimen2*] [*string*]

For the parts of each edge of the source that are not coincident or overlapping with source-compatible figures but are coincident or overlapping with target figures which extend to the interior of the source figure, a rectangle extending normally from the edge out of the source figure by the given dimension is constructed. The test fails if the constructed rectangle is not completely covered by target figures. Figure 15.11 illustrates the test performed under this keyword, for no Region and an *expression* consisting of a single layer.

If the `Enclosed` keyword is given, it requires that where the source and target intersect, the source is entirely covered by the target, with a spacing greater than or equal to the *enc_dimen* between

Figure 15.11: The MinSpaceFrom test. The rule is violated if the projection from the current layer, if any, is less than the supplied dimension.



outside edges. This applies only if the source figure is rectangular. If this clause is used, the *dimesnsion_in_microns* should be set to zero.

The `Opposite` clause also requires that if there is intersection, the source must be entirely covered by the target. This test also applies only when the source shape is rectangular. Two widths are given following the keyword. The test passes if two opposite sides of the source rectangle have extension greater than or equal to the larger of the two numbers, and the other two edges have extensions greater than or equal to the smaller dimension. If the two dimensions are equal, this is equivalent to the `Enclosed` clause. When the two values are different, there is no corner test performed. If this clause is given, the *dimension_in_microns* should be set to zero.

In either clause, a dimension value of 0.0 can be given, meaning that the source and target can share an edge.

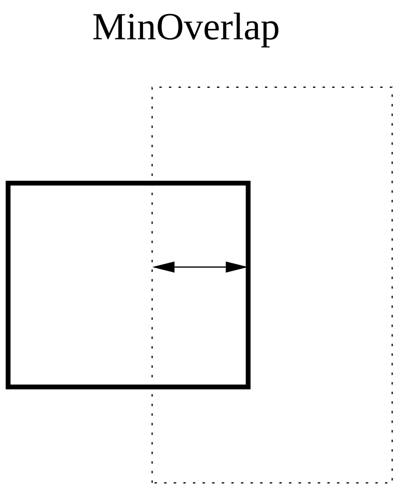
Without the clauses, this is in many cases redundant with the MinNoOverlap test (see below) if applied to the result of the *expression*, if the *expression* is simply a layer name.

The `Enclosed` and `Opposite` clauses are set by `minEnclosure` and `minOppExtension` constraints when importing virtuoso technology data. These may have their own reference strings. To keep these distinguishable, the *string* can actually be three double-quoted strings, e.g., the form is

```
"# rule description" "minEnclosure string" "minOppExtension string"
```

This guarantees that the original reference strings are regenerated when the **!dumpcds** command is used to generate a Virtuoso technology file. If a component string doesn't exist, one can use `""` (two double-quote marks) as a placeholder. Strings to the right that don't exist can be skipped entirely.

Figure 15.12: The MinOverlap test. The minimum width of an intersection of the present layer (solid) and the argument layer (dotted) must not be less than the given dimension.



MinOverlap **Rule**

Syntax: `MinOverlap` [*EdgeArgs*] *expression dimension_in_microns* [*string*]

For the parts of each edge of the source that are not coincident or overlapping with source-compatible figures and are coincident or overlapping with target figures which extend into the interior of the source figure, a rectangle extending normally from the edge into the source figure a distance given by the dimension is constructed. The test fails if the constructed rectangle is not completely covered by target figures. Figure 15.12 illustrates the test performed under this keyword, for no Region and an *expression* consisting of a single layer.

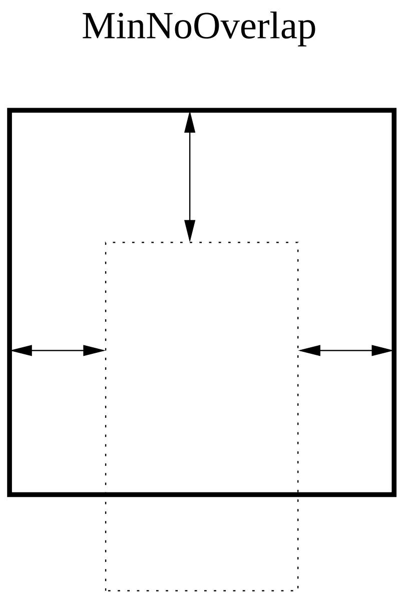
When importing Virtuoso technology data, the `minExtension` constraint maps directly to this rule.

MinNoOverlap **Rule**

Syntax: `MinNoOverlap` [*EdgeArgs*] *expression dimension_in_microns* [*string*]

For the parts of each edge of the source that are not coincident or overlapping with source-compatible figures or with target figures which extend into the interior of the source figure, a rectangle extending normally from the edge into the source figure a distance given by the dimension is constructed. The test fails if the constructed rectangle has nonzero intersection area with target figures. Figure 15.13 illustrates the test performed under this keyword, for no Region and an *expression* consisting of a single layer.

Figure 15.13: The MinNoOverlap test. The minimum width of regions of the present layer (solid) which do not intersect the argument layer (dotted) must not be less than the given dimension.



15.4 Spacing Tables

The design rule checking system supports one and two-dimensional spacing tables, for use in the MinSpace and MinSpaceTo rules. These provide a size-dependent spacing value.

Spacing tables may be imported from Cadence Virtuoso ASCII technology files, or may be created and used exclusively within *Xic*. The format is the same as the *width* and *width—length* spacing tables found in Virtuoso constraint definitions.

In *Xic*, spacing tables are not independent objects, but are owned by a MinSpace or MinSpaceTo design rule. Each rule of these types can have a spacing table. The tables are printed when an *Xic* technology file is generated, as part of the rule specification string. The rule text is printed using backslash line continuation characters for legibility, as the rule string will be rather long when the table is included. The same format is used when listing rules in the **Edit Rules** window, and in the text editor used to edit the spacing table text from the rule editor windows for these rules.

Below is the text for a spacing table, which will be explained as an example.

```
SpacingTable 0.0900 2 0x0 11\
0.0050 0.0050 0.0900\
0.2050 0.3850 0.1100\
0.4250 0.3850 0.1100\
0.4250 0.4250 0.1600\
1.5050 0.3850 0.1100\
1.5050 0.4250 0.1600\
1.5050 1.5050 0.5000\
4.5050 0.3850 0.1100\
```

```
4.5050 0.4250 0.1600\
4.5050 1.5050 0.5000\
4.5050 4.5050 1.5000\
```

The `SpacingTable` keyword begins the definition. Note that each line ends with a backslash character. This "hides" the return character, so that logically the text forms a single line. One could certainly enter the text as a single line, the hidden line breaks are for readability only. The kind and number of text tokens that form the definition are well defined, so there is no ambiguity in where the table definition ends, when it is included in a larger string.

Following the initial keyword are four numeric tokens. The first is the default spacing in microns, as a floating-point number. This will apply when the table does not resolve a value. This should be the same number as the default spacing in the associated design rule, but will override that value if different.

The second number is the table dimensionality, which is either 1 or 2. The example is a two dimensional table, which is probably most common. Each row of a two dimensional table contains three numbers, for *width*, *length*, and the spacing value. In a one dimensional table, the *length* values are not present.

The third number is a hexadecimal value which is used as a flags byte. The flags are saved in tables imported from Virtuoso, and represent unhandled features. The only purpose of these flags is to regurgitate the needed keywords when a Cadence technology file is produced with the **!dumpcds** command. The flags are not documented, but the least significant bit is actually an "ignore" flag used only by *Xic*. The main thing to be aware of is that the spacing table will only be active if the flags integer is 0. If not zero, the table will be carried with the rule, but will not be used in analysis. The default spacing, given in the first number, will apply in any case.

The fourth number is a positive integer giving the number of rows in the table. Each "row" consists of two or three floating-point numbers, for one and two dimensional tables. The row data follow.

The first number in each row is the *width* parameter. In *Xic*, this parameter is obtained from the object whose edges are currently being evaluated for MinSpace or MinSpaceTo violations. If the object is a rectangle, the *width* is the smaller of the rectangle width and height. If the object is a wire, the *width* is the wire's width. If the object is a non-rectangular polygon, the *width* is the smaller of the bounding box width and height.

In two-dimensional tables, the second parameter (*length*) is the parallel run length of the two edges normal to the spacing direction being tested.

The remaining parameter is a spacing value that applies for the given *width*, and *length* in two-dimensionsl tables.

15.4.1 Spacing Table Evaluation

Spacing tables are used only for Manhattan (horizontal or vertical) edges. Non-Manhattan edges will use the Diagional spacing if given, or the default spacing. There is presently no provision for size-dependent spacing of non-Manhattan edges.

The *minspace* dimension will be found in the row where the measured *width* and *length* are greater than or equal to the row *width* and *length*, with the largest spacing value. The evaluation is actually iterative, and follows this logic:

Compute the *width* parameter knowing the object being tested.

Loop over each object edge {

Take the initial *length* to be the total edge length.
Evaluate the spacing table, find the initial *minspace*.
Loop {

Construct a test region along and outside of the edge being tested, with width given by the *minspace*.
Test this region for the presence of target material.
if (none found)
Break, edge test is clean.
Measure the length of each intersection region and sum. This provides a new *length*.
Evaluate the spacing table with the new *length*.
if (the new and old *minspace* are equal)
Break, test indicates violation.
The new *minspace* will be smaller, check again.

}

}

15.5 User-Defined Design Rules

This section describes the facility for defining and referencing user-specified design rules. These allow complex tests to be implemented. User-defined rules are defined in separate blocks ahead of the physical layer specification blocks in the technology file. The rules are referenced from the layer blocks. A user defined rule definition has the following general form:

`DrcTest` *testname arg1 arg2 ...*
`Edge Outside|Inside` *expression*
`MinEdge` *dimension*
`MaxEdge` *dimension*
`Test Outside|Inside` *dimension expression*
`TestCornerOverlap` *dimension*
`Evaluate` *logical_expression*
`[` *script lines* `]`
`End`

The first line of the block starts with the keyword DrcTest. This is followed by a name for the test, which must be unique among the keywords recognized in the technology file. This is the name by which the test will be referenced. Following the name are zero or more argument tokens. These can be any alphanumeric text strings, which represent parameter names. These are the formal arguments to the rule, and appear in the lines that follow in the form "%*token*%", which will be replaced by the actual arguments given in the references to the rule.

The rule is evaluated at each edge of the source. Each edge is divided into segments, depending on specifications. For each segment, a rectangle is constructed, extending either into or out of the source figure. Tests are applied to these regions, The Edge keyword indicates an edge specification. There can be zero or more edge specifications. Following the Edge keyword is one of the keywords Outside or Inside followed by a layer expression. When the edge is evaluated the regions of the edge where the expression is true are found, either just inside or just outside of the figure. The default edge is the set of regions where there is no source figure just outside the edge, which means that there is no source-compatible adjacent figure. The results from each Edge specification are anded together with the default edge to determine the segments where tests are performed. The expression part of the Edge specification can contain argument substitutions.

For example:

```
Edge Inside M2
```

This will include the parts of the figure boundary that 1) do not touch or overlap another figure of the same source (the default edge), and 2) have layer `M2` present on the inside side of the boundary. The default edge is always implicitly included in the conjunction.

The MinEdge and MaxEdge lines, which are optional, allow setting limits on the segments used for testing. If given, an edge segment used for testing would have length greater or equal to the MinEdge dimension, and less than or equal to the MaxEdge dimension. The dimensions appearing after the keywords can contain argument substitutions.

There must be one of more lines given which start with the keyword Test. These specify the tests which are applied to regions constructed from the edge segments. Following Test is one of the keywords Outside or Inside, which determines whether the test area extends outside or inside the source figure. The following token, which can contain an argument substitution, sets the length by which the test area extends out of or into the source figure. The rest of the line contains a layer expression, which can contain argument substitutions, which is evaluated in the test area.

The expression will be evaluated within the test area by one of the evaluation functions described below. If using the most common `DRCuserTest` evaluation function, The test is true if the expression is true somewhere in the test area, meaning that there is a non-zero area where the logical expression would be "dark".

For example:

```
Test Outside 0.5 !M2
```

This test will be "true" if within the rectangle extending out of the figure from the edge by 0.5 microns, there is some point where layer `M2` is not present, if using `DRCuserTest`.

The optional TestCornerOverlap is a special supplemental test when evaluating "`MinWidth`". This measures the mutual edge or overlap of adjacent compatible figures. The width of the mutual edge must be greater than the dimension (which can contain argument substitutions).

The final line, which begins with the keyword Evaluate, specifies a logical expression or script. There are two forms for the Evaluate construct. In the first form, the expression must be cast as an assignment to a variable named "`fail`", and if set true the entire rule fails. Argument substitutions are allowed in the expression. The assignment must appear on the same line following Evaluate.

In the second form, there can be no additional text on the line following Evaluate. The following lines contain a script, in the format understood by the script parser. This is terminated with the keyword EndScript. Argument substitutions are allowed in these lines. The script can contain any of the constructs described in the manual section on the script parser, with the exception of the "preprocessing" directives; any line with a leading '#' is ignored. The script should set a variable named "`fail`" to signal a DRC violation.

There are several functions which can appear in the Evaluate lines. Each of these functions takes a single integer argument. This is a zero-based integer index corresponding to the Test lines, in order of their appearance. Each function returns a value obtained from the corresponding test.

The functions currently available are the following:

(int) `DRCuserTest`(*index*)
The return value is 1 if the test region is not empty, 0 otherwise.

(int) `DRCuserEmpty`(*index*)
The return value is 1 if the test region is empty, 0 otherwise.

(int) `DRCuserFull`(*index*)
The return value is 1 if the test region is completely covered, 0 otherwise.

(zoidlist) `DRCuserZlist`(*index*)
The return value is a list of trapezoids clipped from the test region. The list can be used with script functions that operate with this data type.

(int) `DRCuserEdgeLength`(*index*)
The return value is the length along the edge of the test region. This is the value that is filtered by MinEdge and MaxEdge. Filtered edges will not be seen by this function.

The functions specified are called for each test region for each edge and corner. The return value can be used to set the `fail` variable. Once the `fail` variable has been set nonzero, testing of the object terminates for the present rule.

For example:

```
Test Outside 0.5 !M2
Test Inside 0.5 !M2
Evaluate fail = DRCuserTest(0) | DRCuserTest(1)
```

Here, the test fails if `M2` does not completely cover the area 0.5 microns on either side of the edge. The arguments to the `DRCuserTest` function refer to the `Test` lines: 0 is the first `Test` line in the rule, 1 the second, and so on.

The multi-line variation of the Evaluate clause has the form

```
Evaluate
script line
...
EndScript
```

Within the script, there are a number of predefined variables available. With the exception of `fail`, these all start with an underscore.

`_ObjType`
The type of object which is undergoing DRC. Values are 'p', 'w', or 'b', for polygons, wires, and boxes.

`_ObjNumEdges`
This is the number of vertices in the figure being tested. Boxes and wires are converted to polygons for testing, so this makes sense for all objects. The first and last vertices are the same, and all are counted, so that the number of vertices in a box is five.

`_CurEdge`

This is the zero-based index of the edge or vertex currently being tested. If the original object is a box, the zeroth vertex is the lower-left corner, and the zeroth edge is the left edge. For polygons, the zeroth vertex is the first vertex in the polygon's coordinate list, and the zeroth edge extends from this vertex to the next. This index will cycle through the values from 0 to _ObjNumEdges-1. Values may be skipped of there is no testable area at the edge or corner.

If a test is identified as a "MinWidth" type, i.e., an inside test with the target the same as the source, at most two edges are tested if the figure is a box.

`_CurTest`

This gives the following values: 0 if the test is a standard edge test, 1 if the test if a corner test, and 2 if the test is the CornerOverlap test.

`_CurX1, _CurY1, _CurX2, _CurY2`

These four variables provide the starting and ending coordinates of the edge segment being tested, in microns.

Variables defined within the script remain in scope forever, they do not change between calls.

When an object is DRC tested, the Overlap tests, if any, are first applied to the source region. This is followed by the Area tests, then the edge tests, which include any user-defined tests. During the edge tests, each edge is evaluated in sequence. The test may be applied several times for different regions along the edge or not at all, depending on the geometry and the Edge specification.

Edge segments are evaluated in the order crossed by a point following the boundary starting at the first vertex (lower left corner for boxes). Boxes and wires always have clockwise winding, though polygons can have either clockwise or counterclockwise winding.

Associated with the edge test are the corner tests. For a box, the order of tests is given below. The corner test is applied at each vertex (if indicated by the angle) after the previous adjacent side has been tested. The test area is a polygonal shape designed to "fill in" gaps between the rectangular areas associated with the sides.

_CurEdge	_CurTest	which
0	0	left edge
1	1	upper left corner
1	0	top edge
2	1	upper right corner
2	0	right edge
3	1	lower right corner
3	0	bottom edge
0	1	lower left corner

A rule is implemented by adding a reference to the rule in the layer block of a physical layer. The format is

testname [`Region` *region_expr*] [*arg1 arg2* ...] [*string*]

The *testname* is the keyword defined in one of the rule definitions, as described above. This is followed by an optional source specification, and the actual arguments, which must correspond in number to the rule arguments. These are followed by an optional *string*, which is arbitrary explanatory text.

As initial examples, below are implementations of the built-in rules which involve edge evaluation.

These are the rule definitions, and by convention they appear in the technology file after the electrical layer definitions and ahead of the physical layer definitions.

```
# In the first two rules, lyr is the same as the source
#
DrcTest myMinWidth dim lyr
Test Inside %dim% !%lyr%
TestCornerOverlap %dim%
Evaluate fail = DRCuserTest(0)
End

DrcTest myMinSpace dim lyr
Test Outside %dim% %lyr%
Evaluate fail = DRCuserTest(0)
End

# In the remaining rules, lyr is different from the source
#
DrcTest myMinSpaceTo dim lyr
Edge Inside !%lyr%
Test Outside %dim% %lyr%
Evaluate fail = DRCuserTest(0)
End

DrcTest myMinSpaceFrom dim lyr
Edge Inside %lyr%
Test Outside %dim% !%lyr%
Evaluate fail = DRCuserTest(0)
End

DrcTest myMinOverlap dim lyr
Edge Inside %lyr%
Test Inside %dim% !%lyr%
Evaluate fail = DRCuserTest(0)
End

DrcTest myMinNoOverlap dim lyr
Edge Inside !%lyr%
Test Inside %dim% %lyr%
Evaluate fail = DRCuserTest(0)
End
```

To implement the rules, references are added to the layer definitions:

```
Layer M1
...
myMinWidth 3.0 M1
myMinSpace 2.0 M1
myMinSpaceTo 1.0 M2
...
```

Here are some examples of more complicated rules:

Rule: Objects on M3 smaller that 10 microns must be separated by .5 microns or more, Objects larger than 10 microns must be separated by .75 microns or more.

```
DrcTest myMinSp1 lyr
# Fail if spacing < 0.5
Test Outside .5 %lyr%
Evaluate fail = DRCuserTest(0)
End

DrcTest myMinSp2 lyr
# Fail if spacing < 0.75 and width >= 10
MinEdge 10
Test Outside .75 %lyr%
Test Inside 10 !%lyr%
Evaluate fail = DRCuserTest(0) & !DRCuserTest(1)
End


Layer M3
...
myMinSp1 M3
myMinSp2 M3
...
```

Note that we did not need to use substitution here, as the rule only applies to M3.

Rule: Objects on M3 must be larger than 1 micron, unless over I1 in which case the width must be 1.25 microns.

```
DrcTest myMinW1 lyr
# Fail if width < 1.0
Test Inside 1 !%lyr%
TestCornerOverlap 1
Evaluate fail = DRCuserTest(0)
End

DrcTest myMinW2 lyr
# Fail if width < 1.25 and I1 present
Test Inside 1.25 !%lyr%
Test Inside 1.25 I1
TestCornerOverlap 1.25
Evaluate fail = DRCuserTest(0) & DRCuserTest(1)
End


Layer M3
...
myMinW1 M3
myMinW2 M3
...
```

Rule: The overlap of M1 surrounding Via must be .5 microns or greater. Only two sides maximum can have an overlap of less than 1 micron, the other sides must have 1 micron of overlap or more.

In the script below, two arrays are defined, to hold the test results. We assume that only boxes are used for vias, and ignore the corner tests. When the final edge (_CurEdge = 3) is reached, the results saved in the arrays are evaluated, and the fail flag is set if an error is indicated.

```
DrcTest vtest
Test Outside 1 !M1
Test Outside .5 !M1
Evaluate
tl[4]
ts[4]
if (_ObjType == 'b' & _CurTest == 0)
    tl[_CurEdge] = DRCuserTest(0)
    ts[_CurEdge] = DRCuserTest(1)
    if (_CurEdge == 3)
        if (tl[0] + tl[1] + tl[2] + tl[3] > 2)
            fail = 1
        end
        if (ts[0] + ts[1] + ts[2] + ts[3] > 0)
            fail = 1
        end
    end
end
EndScript
End
```

The test is implemented in the Via layer block. Just the keyword is needed, since no arguments are passed.

```
Layer Via
...
vtest
```

15.6 Assigning Design Rules

Design rules can be added to the technology file by hand with a text editor, or from within *Xic* using the **Edit Rules** pop-up panel in the **DRC Menu**. Note that if macros or the eval construct are to be used in design rules, the text must be inserted with a text editor, as these constructs are unknown to the **Edit Rules** pop-up.

Xic supports a set of design rule primitives which should cover the vast majority of cases encountered in process technology specifications. In addition, more specialized tests can be developed through use of user-defined design rules, described in 15.5.

Most simple specifications translate directly into a rule keyword, in particular, the setting of MinWidth and MinSpace are usually straightforward. MinWidth is generally the smallest feature size allowable on the layer, and MinSpace is the smallest gap allowed between features on the layer. Note that if one feature touches another on the same layer, the tests are applied such that the combined features are measured. Thus it is legitimate to have subdimensional features, as long as they are directly adjacent to other features so that the combined dimensions satisfy the MinWidth test.

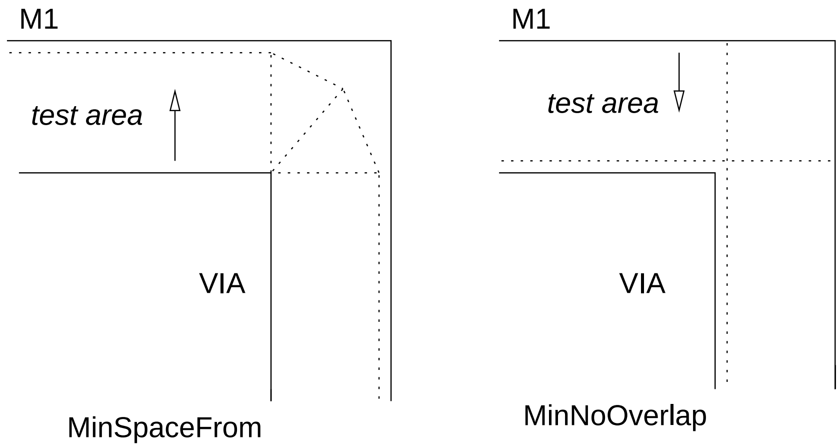
There are six rules which flag overlapping of layers Overlap, IfOverlap, NoOverlap, AnyOverlap, PartOverlap, and AnyNoOverlap. Of these, the first three are by far the most commonly used. If objects on layer A should always be over/under layer B, the Overlap rule should be added to layer A. If objects on layer A should never intersect layer B, the NoOverlap rule should be applied to layer A. The case of coincident layer A and B edges of adjacent objects will not produce an error, unless an additional MinSpaceTo test is applied. If objects on layer A should either be entirely covered by layer B, or not intersect layer B at all, the IfOverlap rule should be added to layer A. In this test, if an object on layer A partially intersects layer B, an error is generated. This is useful for ensuring that a feature does no cross an underlying edge, for example.

In reciprocal rules, such as MinSpaceTo, which specifies the minimum distance between objects on two different layers, it is often questioned whether the rule should be specified in each layer. The answer is no, although no real harm is done if it is specified in both layers, though both specifications had better provide the same dimension. In testing of a newly created object (interactive DRC), first the object is tested with respect to rules defined on the object's layer. If there are no errors, all nearby objects which have a rule target of the object's layer are tested, and any errors are flagged on the new object. For example if a box on layer A is created too close to a box on layer B, and B contains a MinSpaceTo rule with respect to layer A, first the A box is tested (result: ok) then the B box is tested, (result: failure due to proximity to A). The A box is marked and the error region is indicated. In batch mode testing, only the rules for a given object are evaluated. Typically, all objects in the region are tested, so the error will be caught. If there were specifications on each layer, there would be two error messages produced, as two separate but redundant tests would be performed. In the MinSpaceTo test, the condition where edges are coincident is flagged as an error, however if the two objects are actually intersecting with nonzero area, no error is generated from the MinSpaceTo test.

The word "overlap" is confusingly used in two contexts. In process specifications, the "overlap" is often taken as the width of material surrounding a feature, such as a via. In the *Xic* documentation, "overlap" often refers to an area of mutual intersection of two (or more) different layers. As an important example, a process specification might read "overlap of M1 around VIA 1.0 micron". This implies that layer M1 must extend 1.0 microns or more outside of the VIA feature. One way to test this condition is with the MinNoOverlap keyword as a rule on M1: "`MinNoOverlap VIA 1.0`". This specifies that a region along the outside edge of M1 1 micron in width toward the inside of the M1 feature will be checked for the presence of VIA, and an error will occur if any VIA material is found intersecting with this region. The MinNoOverlap test will flag as an error the case where the edges of the two intersecting objects are coincident, however if the VIA area actually encloses M1, no error is generated. The Overlap and IfOverlap keywords can be used to detect this circumstance. Often, the process specification will list such a rule with the interior feature layer (VIA), in which case it makes more sense to use the MinSpaceFrom test as in "`MinSpaceFrom M1 1.0`" applied to the VIA layer. This specifies that a region projecting outward from the VIA feature by 1 micron should be entirely covered by M1. This is almost equivalent to the MinNoOverlap test, however the treatment of the corners is different. This is illustrated in Figure 15.14.

In the case of coincident vias, where the order is not important but the concentric spacing must be greater than some value, mutual MinNoOverlap rules can exist in each layer. In the case where one ordering is prohibited, the Overlap or IfOverlap keywords can be used in the inner layer. For example, suppose VIA1 and VIA2 can be concentric, but VIA1 must be outside of (larger than) VIA2. Layer VIA1 would contain a "`MinNoOverlap VIA2`" directive, layer VIA2 would contain an "`IfOverlap VIA1`" directive if VIA2 can exist independently of VIA1, or an "`Overlap VIA1`" directive otherwise. Then, if VIA2 is larger than VIA1, the partial intersection will trigger an error.

Figure 15.14: The MinSpaceFrom and MinNoOverlap tests differ in the treatment of the corner regions projecting outward from the central feature, as shown.



The MinOverlap test is used to determine whether the intersection width of two layers is larger than some minimum. It is usually used of conjunction with certain types of contacts or vias, to ensure that the contacting area is sufficiently large. The MinArea and MaxArea tests are also useful is this regard. In particular, to test that a via has an exact size (square), a MinWidth and a MaxArea test are both applied. A MinEdgeLength test is used in the circumstance where the edge-crossing width of a layer is larger than the layer's minimum width.

15.7 The Setup Button: Set DRC Limits

The **Setup** button in the **DRC Menu** brings up the **DRC Parameter Setup** panel, which allows the user to set limits and other parameters used in design rule checking.

The top third of the panel provides control of layer and rule filtering. It is sometimes useful to perform design rule checking using only a subset of rules, on only a subset of layers. It may also be useful at times to skip particular rules or layers. The user has this flexibility through the entries in this panel. One also has the ability to inhibit rules individually with the **Design Rule Editor** panel from the **Edit Rules** button in the **DRC Menu**.

At the top of the panel are **Check listed layers only** and **Skip listed layers** check boxes. If either is checked (it is not possible to select both) then the text entry area just below the check boxes becomes un-grayed, and the user is expected to enter a list of layer names, separated by space. Rules on the listed layers will either be used exclusively or ignored during checking, depending on which of the boxes is checked.

The layer filtering entries control the status of two variables. The filtering can also be set up by setting the values of the variables directly.

DrcLayerList
: This variable is set to a space-separated list of layer names, as shown in the text entry area. The variable exists only if there is text shown in the entry area.

DrcUseLayerList

If this variable is not set, then the DrcLayerList variable will be ignored if it exists, and DRC testing will use rules defined on all layers. If the variable is set to a word that starts with 'n' (case-insensitive) or just the letter itself, then the DrcLayerList will be used, if it exists, to provide a list of layers whose rules will be skipped during DRC testing. If DrcUseLayerList is set to something else, including to nothing (set as a boolean), the DrcLayerList, if it exists, will supply a list of layers whose rules will be used during DRC testing. Rules on unlisted layers will not be tested in this case.

Below the layer list entry are **Check listed rules only** and **Skip listed rules** check boxes. These, and the initially grayed text entry area just below, provide an analogous filtering capability based on rule names. The rule names are the names (keywords) of the built-in tests, or the name assigned to a user-defined rule. If ether box is checked (it is possible to check at most one of the boxes), then the text entry area becomes un-grayed and the user should enter a space-separated list of rule names. The name matching is case-insensitive. The listed rules will either be checked exclusively (unlisted rules ignored) or skipped during DRC testing, depending on which box is checked.

The rule filtering entries control the status of two variables. Rule filtering can also be set up by setting the values of the variables directly.

DrcRuleList

This variable is set to a space-separated list of rule names, as shown in the text entry area. The variable exists only if there is text shown in the entry area.

DrcUseRuleList

If this variable is not set, then the DrcRuleList variable will be ignored if it exists, and DRC testing will use all rules that have been defined, and have not been inhibited. If the variable is set to a word that starts with 'n' (case-insensitive) or just the letter itself, then the DrcRuleList will be used, if it exists, to provide a list of rules that will be skipped during DRC testing. If DrcUseRuleList is set to something else, including to nothing (set as a boolean), the DrcRuleList, if it exists, will supply a list of rules that will be used during DRC testing. Unlisted rules will not be tested in this case.

Below the layer and rule filtering group is the limit values group. These provide numeric limit values that are observed while testing. Each of these controls has a corresponding tracking variable (see E.23).

The first limit is on the number of violations reported in batch mode checking. These are runs initiated from the **DRC Run Control** panel obtained from the **Batch Check** button in the **DRC Menu**. If this limit is reached, the checking terminates. Setting this limit (or any of the limits) to zero will inhibit the limiting.

The remaining limits pertain to interactive mode (the **Enable Interactive** button in the **DRC Menu** is active). When enabled, these checks are performed after every operation which modifies the physical geometry in the database. Often, the pause can be quite substantial, and it is preferable to minimize the delay, at the expense of thorough testing. Testing can be performed at a later time using batch mode. The interactive time can be limited in two ways: by limiting the number of objects checked, and by actually setting a time limit. The checking will also terminate when a maximum error count is reached. Of course, interactive testing can be switched off entirely with the **Enable Interactive** button.

The object count limit specifies the maximum number of objects checked per test cycle. The time limit, specified in milliseconds, will terminate testing when the time limit is reached. The error count limit, if nonzero, will terminate testing when the count is reached.

The final choice is a yes/no as to whether to test subcells that are moved, copied, or placed. This is often very time consuming, as all objects in the subcell and its descendents are checked. If the subcell has been checked previously, most of the checking is redundant and can be skipped.

The remaining buttons allow selection of the violation recording level. The default is to record only one violation per object. With many violations, this can cut evaluation time, and may be useful for a first pass. The second choice outputs a maximum of one violation of each type (i.e., corresponding to each rule name keyword) per object. The third choice will output all violations found. This can lead to redundancy, as certain violations may be reported for each *edge* of the offending object.

15.8 The Set Flags Button: Set Skip Flags

The **Set Flags** button in the **DRC Menu** enables the "skip drc" flag to be set or cleared on objects in the current cell. When the flag is set, the object is ignored by the drc tests. Note that this can cause other tests to fail, for example if a subdimensional object is adjacent to another object on the same layer with its skip flag set, the error will be reported. Objects with the skip flag set are shown as selected. The selected objects can be deselected, or other objects selected, in the usual way. In any case, the selected status when the command exits will be represented in the objects' skip flags.

If the layer has the NoDrcDatatype attribute set in the technology file or with the **Edit Tech Params** button in the **Attributes Menu**, objects with the skip flag set will be written with the given datatype rather than the default datatype set in the StreamOut specification, in GDSII and OASIS files.

Objects which intersect a layer named "NDRC" are also skipped during DRC testing. Defining an NDRC layer is an easy way to avoid testing logos, process test features, and other objects which would ordinarily produce many errors.

15.9 The Enable Interactive Button: Set Interactive Checking

When th **Enable Interactive** button in the **DRC Menu** is active, design rule checking is performed on new objects as created, or objects modified, during editing. A violating object is marked, and the error highlighted. A pop-up window explains the violation, unless this has been suppressed with the **No Pop Up Errors** button. The object is included in the database, and the user must decide whether to fix or defer the violation.

This tracks the state of the Drc variable.

15.10 The No Pop Up Errors Button: Suppress Error Report

When the **No Pop Up Errors** toggle button in the **DRC Menu** is set, violations found in interactive DRC do not cause messages to appear in a pop-up window. The error indication is still drawn on-screen, however. The **Query Errors** command can be used to get the error string, or the **Dump Error File** command can be used to obtain a complete report.

This tracks the state of the DrcNoPopup variable.

15.11 The Batch Check Button: Initiate Rule Check

The **Batch Check** button in the **DRC Menu** provides the **DRC Run Control** panel. This panel allows initiation of a batch DRC job, running either in the foreground or background. By "batch", it is meant that the job is not intended to be interactive. Such jobs may take a long time to run.

The panel has two pages, tabbed **Run** and **Jobs**. The **Run** page is generally of most interest and is the default page.

The top two entries in the **Run** page allow a Cell Hierarchy Digest (CHD) to specify the source cells for the DRC run. A CHD is a small database of cell offsets into a large layout file, which allows access to the content of the file without reading it into *Xic*. With limited memory, this allows huge files to be accessed, that would otherwise fail to load due to memory limitation. The first line contains a text entry area into which the user enters the CHD name. This will be ignored unless the **Use** button, to the left of the text entry, is pressed. The second line is optional, and may contain the name of a cell referenced by the CHD. This will be taken as the top-level cell in design rule checking. If no entry is given, the default top-level cell of the CHD is used.

These entry areas mirror the values of the DrcChdName and DrcChdCell variables. The variables are set when the corresponding entry area contains text, unset otherwise. The variables can be set directly, however in any case the **Use** button must be pressed for the values to have any effect during the DRC run.

In order to gain any memory-saving benefit from using a CHD, a partition grid should generally be used. This is set in the third row of the **Run** page. When the partition grid is enabled, the area to test is divided into a grid of the specified size. The DRC testing is performed sequentially for each grid area. In the case of CHD input, only the geometry needed to represent the current grid area is in memory, which should be a small fraction of the geometry of the total test area (if the test area is very small, partitioning is not needed).

To set up a partition grid, un-press the button with the **None** label. This will un-gray the text entry area to the right. This area should be set to the side length in microns of the square grid cell. The size should be small enough so that the geometry needed to represent the area will easily fit in memory, but not so small that the overhead inherent in the gridding increases run time appreciably. The best value is very dependent on the technology, and the user should experiment.

The partition setting entry and button control, and are controlled by, the DrcPartitionSize variable. When this variable is set to a number, that number is taken as the partition grid spacing. The partition size can be set and the use of partitioning enabled by setting this variable directly.

The partition grid also applies when the current cell is the target of the design rule checking. It is not clear at present that the partition grid provides any advantage in this case. Some operations may be performed faster when gridded as opposed to processing the entire layout, due to scaling properties. The use of the partition grid provides something for the user to experiment with.

By default, a DRC test run will test the entire area of the target cell (either the current cell, or the cell implied from the CHD). The group of controls below the **Partition grid size** entries allows the testing area to be set to an arbitrary rectangular window. Of course, the window coordinates should overlap the cell coordinates, or no testing will be done.

When the **Set** button is pressed, a command is active where if the user drags, or clicks twice to define the corners of a rectangle, those values will set the numerical entries on the panel. This can be used to visually set the test area to an area of interest in the displayed layout. This works whether or not the window is actually used, i.e., whether or not the numeric coordinate entry areas are grayed.

The window will be ignored unless the **Use Window** check box is checked. When checked, the numerical coordinate entry areas become un-grayed, and the user can manually enter the window coordinates.

The **Flatten** check box is un-grayed only when the **Use** (use CHD) button is pressed. When checked, as geometry is being read into memory with the CHD, it will be flattened into a single cell. Thus, DRC tests will be applied to a flat cell, which can be more efficient than processing a hierarchy. However, there is overhead in flattening, and the flat representation can take much more memory than a hierarchical representation. This should generally be employed only when using a partition grid or small window. The feature is rather experimental and obscure.

Near the bottom of the **Run** page are large **Check** and **Check in Background** buttons. These will launch a DRC run in the foreground or background. The run will observe the other settings in the panel (and in the **DRC Parameter Setup** panel).

When a job is started in the foreground with the **Check** button, the button remains pressed until the job completes. If the user un-presses the button, the job will be paused, and can be terminated by the user. The job can also be paused by pressing **Ctrl-c** when *Xic* has keyboard focus. Other operations are locked out while DRC is running. Violations are recorded in a file named `drcerror.log.`*cellname* which is written in the current directory. Additionally, if not using a CHD, violating objects are marked, and the error region highlighted.

When a foreground job is running, *Xic* is busy and unusable. When a job is started in the background, however, *Xic* is available for other tasks. There can be multiple spawned processes executing concurrently. A pop-up window will appear alerting the user that a job has completed.

Unlike the foreground run, violations are not marked on-screen. The **Update Highlighting** button can be used to generate the highlighting after a background run completes. If the **Show Errors** mode is active, and the current cell is the same as that being checked, when a background job terminates, the error display window is popped down and the mode terminates.

The spawned process is set to ignore the `SIGHUP` signal, so that the process will continue to run if the user's shell is destroyed and/or the user logs out. This is the preferred method by which large, batch DRC jobs can be performed.

This process will create an errors file in the current directory named `drcerror.log.`*cellname*`.`*PID* where *PID* is the process id of the spawned process.

Under Windows, this works by executing a batch-mode *Xic* process in the background. Presently, this doesn't allow background jobs to use a CHD or partitioning.

The **Jobs** page provides a list of background jobs currently running. Jobs can be selected by clicking on the text. When a job is selected, the **Abort job** button becomes un-grayed. Pressing this button will halt the selected job. Be aware that there is no confirmation, and it is not possible to restart a job that is halted in this manner. The spawned process can also be stopped or killed using the job control functionality of the user's shell.

15.12 The Check In Region Button: Check Objects

When the **Check In Region** button in the **DRC Menu** is active, design rule checking is performed on objects the user clicks on or drags over. The method of selecting a region to check is the same as for the **Check In Foreground** command, however the **Enter** key is ignored. Violating objects are marked, the error region highlighted, and a pop-up explains the error. No file is produced. A maximum of 15 errors are accumulated for each region — the check terminates at this error count. The **Check In Region** command button remains active until explicitly terminated, unlike the **Check In Foreground** command.

15.13 The Clear Errors Button: Clear Error List

Pressing the **Clear Errors** button in the **DRC Menu** will delete the internal list of error-producing objects, and consequently clear the display of highlighting and error boxes associated with violations. This does not affect to objects in the database.

15.14 The Query Errors Button: Print Error Text

When the **Query Errors** button in the **DRC Menu** is active, clicking on the highlighted error region (not the object, but the highlighted figure which indicates the location of the error) will display the text of the error message for that error on the prompt line.

15.15 The Dump Error File Button: Save Errors to File

The **Dump Error File** button in the **DRC Menu** allows the user to dump a file containing the error records for the currently visible (as highlighting) errors.

The user is first given the chance to provide the file name, which should begin with "`drcerror.log`" to be recognized as a DRC errors file for subsequent reading into *Xic*. After the file is created, the user is given the option to view the file in a **File Browser** window. If no file name is given, the file will be written to a temporary file which is erased on program exit. This may be convenient if the user only wants a quick view of the text.

Tha **Update Highlighting** command button provides the reverse operation, recreating the highlighting from an existing error log file.

15.16 The Update Highlighting Button: Create Highlighting from File

The **Update Highlighting** button in the **DRC Menu** will delete the internal list of DRC error highlighting indicators, and rebuild the list from a DRC error log file. The error log file must exist in the current directory, have a file name beginning with "`drcerror.log`", and apply to the current cell. If there are multiple files found, a listing will appear, allowing the user to make a choice. After selecting an entry, pressing the **Apply** button on the list pop-up will continue the operation.

DRC error files are produced by the batch mode checking operations initiated from the **DRC Run Control** panel. In the foreground check, the highlighting list is generated along with the file, however no highlighting is produced in background checking, so this command can be used to visualize the errors in that case. It can also be used to bring back the highlighting that was cleared with the **Clear Errors** command, if there is a corresponding error log file.

The **Dump Error File** command performs the reverse operation, creating an error log file from the internal highlighting list.

The **!errs** prompt line command performs the same operations as this button.

15.17 The Show Errors Button: Show Next Error

After batch rule checking (using the operations initiated from the **DRC Run Control** panel) is performed, or in any case when a compatible DRC error log file is present in the current directory, errors from the file may be graphically viewed sequentially with the **Show Errors** button in the **DRC Menu**.

When the **Show Errors** button is pressed, if there is only one error log file for the current cell, it is loaded, otherwise a list of files is presented and the user must make a selection, then press the **Apply** button. If a file is successfully loaded, the **Show Errors** button in the menu will be shown active, and a message will appear in the prompt area. The search for error files extends only to the current directory, and only to files with a name beginning with "`drcerror.log`". The file must have been generated from a cell with the same name as the current cell.

This sets a mode where pressing the **PageDown** key will display the first and subsequent errors in a sub-window. The **PageUp** key can be used to view previously displayed errors. The **Ctrl-f** key performs the same operation as **PageDown**, and the **Ctrl-b** and **Ctrl-p** keys are equivalent to **PageUp**.

The **PageDown** or **Ctrl-f** keys can be used to access the errors randomly, by number. Entering a number followed by **PageDown** or **Ctrl-f** will display the corresponding error. One can also enter + or − ahead of the number, in which case **PageDown** and **Ctrl-f** will move backward or forward in the list by the number.

The functionality is maintained until the **Show Errors** button is selected a second time, making it inactive, or the sub-window is dismissed. The mode *cannot* be exited with the **Esc** key. Any command can be executed when the **Show Errors** button is active, making it possible to interactively fix the errors without leaving **Show Errors** mode.

If a DRC background run terminates when the **Show Errors** mode is active, and the checked cell is the same as the current cell, the error display window will be popped down, and the mode exited. The mode can be restarted to view the errors from the new file.

Note that in the sub-window, only the current error is highlighted, whereas in other windows, all errors may be highlighted, if a highlighting list exists. The highlighting list can be created or rebuilt from the file with the **Update Highlighting** button.

Show Errors mode is terminated if a new cell is opened for editing, including **Push** and **Pop**, and upon switching to electrical mode.

15.18 The Create Layer Button: Create Error Region Layer

The **Create Layer** button in the **DRC Menu** will create objects on a given layer corresponding to the error regions in the current highlighting list. These are the actual error regions with solid outline highlighting, and not the "bad" objects which are also marked but with a dashed outline. This operation can be useful for adding the errors to a design file for subsequent processing, and for other purposes.

The **Update Highlighting** command button can be used to generate a highlighting list from an existing DRC error log file.

The user is first prompted for a layer name. Any suitable layer name can be given. A new layer will be created if the name does not match an existing name.

The layer will be cleared before the operation starts. Objects (database polygons and boxes) will be created only in the current cell.

A second prompt allows the user to provide an integer property value. If the user supplies a value larger than 0, a property will be applied to each object with the given number, containing a string with the text of the corresponding error message. The **Show Phys Properties** mode, available from the **Main Window** sub-menu of the **Attributes Menu** and the **Attributes** menu of sub-windows, can be used to display these messages. If a positive integer is not given, no property will be stored with the new objects.

The **!errlayer** prompt line command performs an identical operation.

15.19 The Edit Rules Button: Rule Editor Panel

The **Edit Rules** button in the **DRC Menu** brings up the **Design Rule Editor** panel. The editor contains a listing of design rules for the current layer. The rules for any layer can be displayed by clicking in the layer menu and selecting a new layer. Design rules for the current layer can be added, deleted, modified, or disabled. The **Save Tech** command in the **Attributes Menu** can be used to write a new technology file in the current directory that reflects the changes made.

The rules are listed one per line, using the same syntax as the specification in the technology file. Rules are shown after any technology file macros have been expanded, and macros can not be used in new rules entered from the **Design Rule Editor** panel. Clicking with button 1 on a rule will cause it to become selected (or deselected if it was already selected). Selected rules are acted on by the **Edit**, **Delete**, and **Inhibit** commands in the **Edit** menu of the **Design Rule Editor** panel. The selected rule is shown highlighted.

The **Quit** button in the **Design Rule Editor** panel **Edit** menu retires the rules panel. This can also be accomplished by pressing the **Edit Rules** button in the **DRC Menu** a second time.

If a rule is selected, pressing the **Edit** button in the **Edit** menu will cause the **Design Rule Parameters** entry panel to appear if not already visible, from which the rule parameters can be modified. Once parameters are modified, the **Apply** button will make the changes and update the listing, and dismiss the panel. The rule most recently edited can be reverted to the previous parameters with the **Undo** button in the **Edit** menu of the **Design Rule Editor** panel.

If a rule is selected, pressing the **Delete** button deletes the rule from the current layer. The most recently deleted rule can be restored with the **Undo** button. Deleted rules are really gone, and will not be written to the technology file during update.

The **Inhibit** button toggles the inhibited status of the rules. An inhibited rule is listed with an 'I' in the first column, and is not applied when checking is performed. It is useful on occasion to temporarily disable a rule. The **Save Tech** command will write all rules present to the technology file, inhibited or not. The inhibited status is active only for the current *Xic* session.

If a rule is selected, pressing the **Inhibit** button will change the inhibited state of the selected rule, and deselect the rule.

The **Undo** button undoes the last edit, addition or delete operation. A second press will redo the undo.

The **Rules** menu bar item produces a drop-down list containing the names of the built-in design rules. Selecting a button will cause the **Design Rule Parameters** entry panel to appear if not already visible, from which the rule parameters can be entered. Once parameters are entered, the **Apply** button will add the rule and update the listing, and dismiss the panel. A new rule will replace an existing rule of the same type and target layer. There is also an entry which allows references to user-defined rules to be created. If user-defined rules have been defined, the entry produces a sub-menu of the defined rules. Selecting one allows instantiation on the current layer.

The **Rule Block** button produces a drop-down menu containing the names of existing user-defined rules, plus entries **New**, **Delete**, and **Undelete**. Selecting one of the rule entries brings up a text editor window loaded with the rule block text. The text can be modified, and when saved the internal rule will be updated. This will be reflected in the technology file created with the **Save Tech** button in the **Attributes Menu**. Selecting the **New** entry will open an empty editing window, into which the text of a new rule can be inserted. Saving the text adds the new rule to the internal list.

To delete a user-defined rule, press the **Delete** button in the **Rule Block** menu, then select a rule from the same menu. That rule will be removed from the menu. The rule can be restored with the **Undelete** menu entry, but only one deletion is remembered. When a rule block is deleted, all instances of the rule (in the layers) are inhibited, but not deleted. They are cleared when the internal backup copy of the deleted rule is deleted, which happens on the next rule deletion or when the pop-up is dismissed. If a rule is undeleted, its instances are uninhibited.

When a user-defined rule is edited and saved, the instances of the old rule (of the same name) are inhibited, but are not cleared. The old rule instances are left as an indication of where the previous rule was applied and what arguments it takes. To apply the new rule, the old instances should be deleted by hand, and a new instance created. If the inhibited rules are uninhibited from the menu, the old rule will be used, not the new one. If a technology file is created with the **Save Tech** command, the inhibited rules will be included, so that it is important to delete these if the call to the new rule is different from the call to the old.

15.19.1 The Design Rule Parameters Panel

This panel, which is polymorphic and specific for each design rule type, appears when a design rule is edited or a new rule is being created from the **Design Rule Editor** panel. It provides the appropriate entry areas for rule parameters. If the target rule is changed while the panel is visible, the panel will reconfigure itself to provide the entries for the new rule.

There are entries that are common to multiple rules. All rules have an entry labeled "**Decsription string**", which contains optional arbitrary text which explains the rule or provides a reference. This text will appear in violation messages. All rules but Exist contain an entry labeled "**Layer expression to AND with source figures on current layer (optional)**". This is the optional Region specification. In addition, the "edge" rules contain two entries: "**Layer expression to AND at inside edges when forming test areas (optional)**" and similar for outside edges. These can provide values for the optional Inside and Outside keywords.

The entry areas for the rules are briefly described below. See the rule descriptions for more information.

`User Defined Rule`

User-defined rule arguments (*n* required)
An entry area where the rule arguments are entered, separated by space. The label prints the

number of arguments required for the rule, extra arguments are ignored.

`Connected`
No additional entries.

`NoHoles`

Minimum area (square microns)
If larger than 0.0, holes with an area smaller than this value will trigger an error.

Minimum width (microns)
If larger than 0.0, holes with a width less than this value will trigger an error.

If both of these parameters are 0.0, any hole will trigger an error.

`Exist`
No additional entries.

`Overlap`
`IfOverlap`
`NoOverlap`
`AnyOverlap`
`PartOverlap`
`AnyNoOverlap`

Target layer name or expression
This is the name of a layer, or a layer expression, which is the target for the rule. An entry is mandatory.

`MinArea`

Minimum area (square microns)
This specifies the minimum area for the rule.

`MaxArea`

Maximum area (square microns)
This specifies the maximum area for the rule.

`MinEdgeLength`

Target layer name or expression
This is the name of a layer, or a layer expression, which is the target for the rule. An entry is mandatory.

Minimum edge length (microns)
This specifies the minimum edge length for the rule.

`MaxWidth`

Maximum width (microns)
This specifies the maximum width for the rule.

`MinWidth`

Minimum width (microns)
This specifies the minimum width for the rule.

Non-Manhattan "diagonal" width
If nonzero, this value will be used instead when the measurement direction is not parallel to the x or y axis.

`MinSpace`

Default minimum spacing (microns)
This specifies the default minimum space for the rule.

Non-Manhattan "diagonal" spacing
If nonzero, this value will be used instead when the measurement direction is not parallel to the x or y axis.

Same-Net spacing
If nonzero, this value will be used instead when the measurement is between objects in the same wire net. **This is currently not implemented.**

Use spacing table
When checked, a spacing table (see 15.4) will be used. This provides minimum space based on the source width and running parallel overlap length. The table is consulted for Manhattan edges only. The table can be created or edited with the **Edit Table** button, which brings up a text editor window containing any existing table.

`MinSpaceTo`

Target layer name or expression
This is the name of a layer, or a layer expression, which is the target for the rule. An entry is mandatory.

Default minimum spacing (microns)
This specifies the default minimum space for the rule.

Non-Manhattan "diagonal" spacing
If nonzero, this value will be used instead when the measurement direction is not parallel to the x or y axis.

Same-Net spacing
If nonzero, this value will be used instead when the measurement is between objects in the same wire net. **This is currently not implemented.**

Use spacing table
When checked, a spacing table (see 15.4) will be used. This provides minimum space based on the source width and running parallel overlap length. The table is consulted for Manhattan edges only. The table can be created or edited with the **Edit Table** button, which brings up a text editor window containing any existing table.

`MinSpaceFrom`

Target layer name or expression
This is the name of a layer, or a layer expression, which is the target for the rule. An entry is mandatory.

Minimum dimension (microns)
This specifies the minimum projection for the rule.

Dimension when target objects are fully enclosed
If nonzero, this value will be used to test objects that are fully surrounded.

Opposite side dimensions
If at least one of the two numbers is nonzero, these will be used to test fully enclosed boxes. Two opposite sides must be enclosed by at least one value, and the other two sides must be enclosed by at least the other value.

`MinOverlap`

Target layer name or expression
This is the name of a layer, or a layer expression, which is the target for the rule. An entry is mandatory.

Minimum dimension (microns)
This specifies the minimum overlap width for the rule.

`MinNoOverlap`

Target layer name or expression
This is the name of a layer, or a layer expression, which is the target for the rule. An entry is mandatory.

Minimum dimension (microns)
This specifies the minimum projection for the rule.

Chapter 16

The Extract Menu: Extraction and Verification

Xic contains a facility for extracting a netlist from the physical database, and comparing it with the schematic in the electrical database. *Xic* can recognize devices in the physical layout, extract geometric and electrical data from these devices, and correspondingly update properties of electrical device instances. The netlists extracted from the physical and electrical databases can be compared. This layout vs. schematic (LVS) testing is a useful means of minimizing mask errors.

The **Extract Menu** contains command buttons for performing extraction and related functions. The commands are summarized in the table below, which provides the internal command name and a brief description.

Extract Menu			
Label	**Name**	**Pop-up**	**Function**
Setup	`excfg`	**Extraction Setup**	Set up and control extraction
Net Selections	`exsel`	**Path Selection Control**	Select groups, nodes, paths
Device Selections	`dvsel`	**Show/Select Devices**	Select and highlight devices
Source SPICE	`sourc`	**Source SPICE File**	Update from SPICE file
Source Physical	`exset`	**Source Physical**	Update electrical from physical
Dump Phys Netlist	`pnet`	**Dump Phys Netlist**	Save physical netlist
Dump Elec Netlist	`enet`	**Dump Elec Netlist**	Save electrical netlist
Dump LVS	`lvs`	**Dump LVS**	Save physical/electrical comparison
Extract C	`exc`	**Cap Extraction**	Extract capacitance using Fast[er]Cap
Extract LR	`exlr`	**LR Extraction**	Extract L/R using FastHenry

In addition to the commands available from the **Extract Menu**, the extraction system provides a number of prompt-line commands which provide additional or supplemental capability. These include the **!antenna** command for testing the antenna effect on wire nets connected to MOS gates, and the **!netext** command for batch extraction of physical wire nets from a layout.

16.1 Extraction System: Methodology and Overview

To use the extraction capability, one typically first designs the circuit in electrical mode, producing a schematic, which operates correctly in simulation. One then produces a corresponding layout in physical mode. Generally, the objective and requirement is that the layout vs. schematic test initiated with the **Dump LVS** button in the **Extract Menu** will show no errors. This usually requires that the subcircuits, if any, individually pass LVS. Thus, one would build the cell hierarchy from the bottom up, enforcing LVS to pass at each level.

The *Xic* extraction system differs from others in that it does not (at least presently) provide fixed associations between electrical and physical devices, and subcells. The user has the ability to place cell contact terminals in the layout, and to create net name labels in the layout and set schematic net names by various means, and matching names will associate. Many cells, though, will associate with no user intervention. If the association initially fails, generally a placed cell terminal or two, or judicious use of net name labels, will allow correct association and passing of LVS testing.

There are three important diagnostic and setup tools available from the buttons at the top of the **Extract Menu**. The **Setup** button will bring up the **Extraction Setup** panel. This panel contains four tabbed pages. The **Views and Operations** page contains controls that make certain extraction-related features, such as terminals and group numbers, visible. It also allows terminal placement and parameter editing. It provides means to clear and run extraction, which is otherwise automatic. There is also provision to select devices and other objects that fail to associate. The other three pages of this panel provide controls which map to variables and flags which control extraction system behavior. Most of this will be set in the technology file, and it is unlikely that there will be a frequent need to change the parameters interactively. The **Misc Config** page does provide some display attribute choices which may be an exception.

The **Path Selection Control** panel is obtained from the **Net Selections** button in the **Extract Menu**. This allows visualizing conductor groups, and the corresponding nets in the schematic if there is an association. One can click to select the groups/nets, or enter a group number to highlight that group. The highlighting can follow the net as it descends through the cell hierarchy to any depth. The panel provides a very useful tool for diagnosing net connectivity problems.

The **Show/Select Devices** panel appears on pressing the **Device Selections** button in the **Extract Menu**. This will list the devices found in the current physical layout. These devices can be highlighted by index number, or selected by clicking on them. When selected, measured parameters may be printed and/or compared with the dual device in the schematic. The panel is a useful tool for addressing device recognition issues.

There are cases where one starts with a layout, and it is desirable to generate a schematic. There are also situations where the physical and electrical designs are generated in separate files, and it is desirable to merge these into a single file. *Xic* has provisions to assist in these cases.

Schematics can be generated in various ways. The schematics that are machine-generated by *Xic* have each device individually connected to gnd or tbar terminals, so there are no wires. These schematics are electrically correct, but lack human-readability and aesthetics. They serve, however, as a starting point if the user wishes to rearrange the devices and add wires as in a normal schematic.

A schematic can be generated from a SPICE file with the **Source SPICE** button in the **Extract menu**. This can create devices and subcircuits as needed. Existing devices will have properties updated with values from the SPICE file.

Similarly, the **Source Physical** button will create or update the schematic from an intermediate SPICE file extracted from the physical layout. Existing devices will have properties updated with values extracted from the physical layout, and missing devices and subcircuits are added.

The **Import Control** panel from the **Convert Menu** is used to copy either the electrical or physical part of another cell into the current cell. It is able to extract this information from cell definitions within an archive file. This can be used to combine separate electrical and physical designs into a single hierarchy.

Separate commands are available for generating netlist files from the physical and electrical data. The **Dump LVS** command in the **Extract Menu** performs the layout vs. schematic comparison, and prints errors in a file which may be displayed on-screen.

The **Dump Phys Netlist** command in the **Extract Menu** generates a connectivity listing extracted from the physical database. This includes a listing of extracted devices, in various formats. One format is SPICE, so that the **Dump Phys Netlist** command can be used to generate a SPICE listing extracted from the physical layout.

Commands in the **Extract Menu** also work with the node mapping facility for SPICE output. It is often necessary to know the name of specific circuit nodes in a SPICE file, which by default is not possible as *Xic* assigns then internally. The node mapping facility, controlled with the **nodmp** button in the electrical mode side menu, allows the node tokens to be preassigned.

16.2 Extraction System: Logging and Error Reporting

If, while running the grouping, extraction, or association operations, the operation cannot complete due to an error, or if an important error is identified such as problems in the technology file setup, a file browser window containing the `extraction.errs` file will appear. This file is created in the log files area, and will contain messages indicating serious or fatal errors encountered during processing. When such an error occurs, the file will be made visible automatically, so there is usually no reason to explicitly view this file in the absence of any error indication.

Note that lack of successful association is not considered an error. It is up to the user to make sure that the electrical and physical designs are consistent, and that terminals get placed correctly.

Logging of these operations can be enabled from the **Logging Options** panel from the **Logging** button in the **Help Menu**. The **Grouping/Extraction/Association** group in the lower half of the panel contains four check boxes: **Group**, **Extract**, **Assoc**, and **Verbose**. Checking any or all of the first three will enable logging of the checked operation, producing log files named `group.log`, `extract.log`, and `associate.log` in the log files area. If the **Verbose** box is checked, the files will contain additional information. These (and all) log files are accessible through the **Log Files** button in the **Help Menu**.

The log files may be helpful to the user or to Whiteley Research in resolving problems. The formats are not documented, but are intended to be reasonably suggestive if not self-explanatory to an advanced user.

16.3 Extraction System: Operations and Algorithms

There are three internal operations performed by the extraction system: grouping, extraction, and association. These operations are performed as needed, and have to be performed only once, unless the cell is modified. This accounts for the delay and activity noted in response to initiating many of the command buttons and other operations related to the **Extract Menu**. The technology file specifies the information necessary to perform these operations.

The extraction subsystem requires that a number of items be set up properly in the technology file. This includes the setting of keywords in layer blocks to identify layers that serve as conductors or vias, and definition of device blocks which allow certain devices to be recognized by their physical structure. Wire nets and subcircuit connectivity are determined automatically by assembling groups of similar objects that touch or overlap, or are connected through a via. Once conductor groups are established, devices are extracted, and device terminals are assigned a conductor group index. This results in a description of the circuit which can be compared with the electrical schematic for consistency.

In the present version, all device extraction is performed automatically, thus there is no need or provision for manual placement of device terminals. The connection terminals to the current cell are created in the schematic with the **subct** button in the side menu. It is possible to manually place the corresponding physical cell terminals in the layout with the **Edit Terminals** button in the **Views and Operations** page of the **Extraction Setup** panel from the **Setup** button in the **Extract menu**. However this is not always necessary, as *Xic* will attempt to place the terminals at a correct position in the layout automatically. If this fails, manually placing terminals will assist in the association process, and may be needed in some cases to resolve ambiguity.

Many of the devices in the device library have physical terminal extensions in their node data structures. These are the "physical" devices, such as resistors, capacitors, and transistors. Other devices, such as voltage and current sources, are not physically implementable and have the nophys property assigned. These devices have no physical terminal extensions, thus will not appear in netlists generated from the physical layout. There is a third class of "device" in the device library, which includes the gnd (ground) and tbar terminals. These have no explicit physical implementation, but are an implicit part of the wiring net as they assign connections by name to locations in the schematic. All points connected to terminal devices with the same name are logically connected together. The terminal devices include multi-contact ("bus") terminals, which again pertain to the schematic only and have no counterpart in the layout.

When a device is placed in the electrical schematic, a physical terminal for each device connection is associated with the physical cell. In physical mode, these terminals are made visible with the **All Terminals** and **Cell Terminals Only** check boxes in the **Show** group in the **Views and Operations** page of the **Extraction Setup** panel. Before association, these terminals are grouped just outside of the lower left corner of the physical cell's bounding box. During association, these terminals are automatically moved to their proper locations in the physical layout, if the corresponding physical device structure is correctly identified and associated. One can select and investigate physical devices in the layout with the **Show/Select Devices** panel from the **Device Selections** button in the **Extract Menu**.

Separate commands are available in the **Extract Menu** for generating netlist files from the physical (**Dump Phys Netlist**) and electrical (**Dump Elec netlist**) data. The **Dump LVS** command performs the layout vs. schematic comparison, and prints differences in a file which can be displayed on-screen.

16.3.1 The Grouping Operation

Initial geometric processing is performed, such as implementing the `Conductor Exclude` definitions. The extraction system maintains a shadow cell database. The cells in the extraction database may contain modified objects and flattened subcell geometry, different from the cell in the main database. The extraction database is created before grouping. If the **Extraction View** check box in the **Show** group in the **Views and Operations** page of the **Extraction Setup** panel from the **Extract Menu** is checked, the physical drawing windows will display the cell content based on this database, and not the main database.

One complexity that arises is that a device such as an inductor or transmission line is often implemented simply as a strip of conducting material. In order to insert the device, the grouping algorithm has to be fooled into thinking that the strip which is the device is actually two disconnected strips, one for each terminal. This can be accomplished with the introduction of special layers used in layout, and the `Conductor Exclude` directive. The directive will logically remove parts of the conductor that intersect with the special layer. Perhaps a more familiar example is a MOS transistor, whose body is defined by POLY over ACTIVE. Both are required to be conductors, but without "`Conductor Exclude POLY`" in the ACTIVE layer definition block, the source and drain would be shorted together!.

In grouping, conducting nets are identified. Every conducting object is given a group number. The group number is the same for all of the conducting objects in a wire net, and each disjoint wire net has a unique group number.

Grouping is done recursively, starting from the leaf subcells and working up to the current cell. The core of the grouping is an algorithm for determining conductor paths due to touching objects on a layer, and through vias between layers if the `Via` keyword has been included in the technology file for the via layers, and by contact layers if the `Contact` keyword was applied in the technology file. Once a cell is processed, it is not regrouped unless the cell is modified. The **Groups** check box in the **Show** group in the **Views and Operations** page of the **Extraction Setup** panel can be checked to display the group numbers of objects in the layout in the drawing windows. Each group (conductor net) is assigned a number. While the **Groups** check box is checked, these numbers are printed on-screen near the conducting objects.

16.3.2 The Extraction Operation

Extraction is the identification of physical devices and subcircuits, and establishment of the connections between them. This is the most compute-intensive and time consuming part of the process. Extraction is done recursively, starting from the leaf subcells and working up to the current cell. It requires that the grouping operation has been performed and the group numbering is up to date.

Initially, subcells that should be flattened into the cell are identified, and the flattening performed. On the initial pass, cells that are wire-only, i.e., contain no devices and only wire-only subcells, and cells that have been explicitly specified as flattenable by the user (see 16.4), will be flattened. In flattening, objects from the master cell are transformed and added to the containing cell, replacing the cell instance. This is done in the shadow cell database used by the extraction system. The extraction may be repeated for a cell, if it is determined subsequently the additional subcells require flattening. This may not be known until the association stage. Suffice it to say that the process is iterative and a bit more complicated than the simple progressive flow implied in this description.

Vias and similar wiring cells should have no electrical terminals, and should not be placed in the schematic.

The extraction operation will look for net labels and physical cell terminals that have been placed by the user. These, if found, supply text names for the group over which they reside. The names are saved along with the object list and other parameters for each conductor group, for later use.

In extraction, devices are recognized by the patterns specified in the device definitions in the technology file. The device contact points are identified, and connecting conductor group numbers recorded. Connections to and between subcells are identified and recorded.

In each subcircuit instance, each conductor group is extracted, transformed to parent cell coordinates, and compared with the parent conductor groups and other subcircuit conductor groups for connectivity. Connectivity between conductor groups can be established through

1. similar `Conductor` or ground plane layers touching.
2. `Contact` and `Conductor` or ground plane layers touching.
3. an area of a `Via` layer exists, at any level of the hierarchy, under which the two via layers exist (one from two different groups) and any conjunction expression is true.

Conductor groups are merged when necessary due to being connected through subcircuits, flattened and not. When two groups merge, the object lists are merged, and the larger of the two group numbers is replaced by the smaller. When extraction is finished, the groups are renumbered so that the numbering is compact.

16.3.3 The Association Operation

The association algorithm logically links devices, subcircuits, nets, and terminals between the schematic and layout. If association is successful, then LVS will pass.

At its core, the association algorithm works by comparing candidate similar objects from the schematic and layout, and computing a numerical score. The pair with the highest score "wins" and the two objects become duals of one another, i.e., become "associated". This is done for devices, subcircuits and nets/groups.

Unfortunately, reality is not that simple, and the actual association algorithm is quite complex. Some of the factors contributing to the complexity are listed below.

- The hierarchy tree may not be quite the same in the schematic and layout. The association algorithm has provision for detecting when necessary and logically flattening both the schematic and the layout.
- The circuit may have topological symmetry, where the comparison test fails because the top score is shared by two or more pairs. The algorithm will try various ways to break the symmetry, and if that fails, a random choice will be made. Finding the correct permutation in this type of case requires examination of the context of instances of the cell in the hierarchy.
- The layout may have split nets, where a logical net (as shown in the schematic) consists of two or more disjoint conductor groups in the layout. Instances of the cell have the disjoint conductor groups connected by metal from outside of the master cell. The association algorithm has provision for detecting and accommodating this case.
- The layout and schematic may show different connections to permutable subcell terminals, such as permutable inputs to a logic gate. The association algorithm attempts to detect this type of case and avoid flagging it as an LVS error.

Association is done in (at least) two passes. The first pass is done to the complete cell hierarchy from the bottom up, and corrections that involve comparison at different levels of the hierarchy are skipped. Each cell is associated as far as possible, with no attempt to break symmetries.

On the second pass, inter-hierarchy corrections are allowed (since the parameters to be compared have now presumably been set), and symmetry-breaking is allowed.

During the first pass, a list of symmetries is generated if association fails to complete due to symmetry. If this list is found in the second pass, "symmetry trials" will be initiated. Each symmetry trial represents one choice, or permutation, of the symmetries. Association proceeds with consistency tests applied. If a consistency test fails at a later iteration, the present symmetry trial is aborted, and all associations made in the trial are undone. A new symmetry trial, using a different permutation, begins. Eventually, unless limits are exceeded, the "correct" permutation will be found and association will complete without errors. The MaxAssocLoops and MaxAssocIters variables set the limits. The MaxAssocLoops is approximately the maximum number of permutations that can be accommodated. The MaxAssocIters variable sets the maximum number of calls to the comparison functions until no further associations are found. Only strange cases, such as long series arrays of identical devices, require more than a few iterations.

The **Misc Config** page of the **Extraction Setup** panel from the **Extract Menu** provides a number of controls affecting association.

16.4 Extraction System: Cell Hierarchy and Flattening

When associating, *Xic* will in most cases correctly account for differences in the cell hierarchy in electrical and physical modes. This is most often automatic, though it is possible for the user to intervene if necessary. The following examples illustrate hierarchy differences.

- An instantiated device pcell, where the physical part is represented by a subcell (a pcell sub-master). This differs from normal devices, which have an empty physical part. In this case, the physical subcell will need to be logically flattened into its container cell.
- The schematic might contain symbols such as logic gates, where the gates are implemented with transistors in the physical counterpart cell. In this case, the gate schematic must be logically flattened into the parent schematic.
- Devices and subcircuits shown in the schematic might be found in a subcell of the physical counterpart. There may be a container cell containing logic gates, for example, which itself has no schematic, and the logic gates appear in the schematic. In this case, the container cell will need to be flattened into its container.

Physical cells that are wire-only, i.e., contain no devices or non-wire-only subcircuits, are always flattened into their container. Physical cells that are pcell instances will always be flattened. Physical and electrical cell instances will be flattened as needed during extraction. Thus, there is typically little need for the user to set up explicit cell flattening, much less so than in earlier *Xic* releases. However, the methods for explicit flattening are still available and can be used if *Xic* finds a flattening situation that isn't handled properly automatically.

When a physical subcell instance is "flattened", all conducting groups, devices and sub-subcells in the subcell master are transformed and linked into the containing cell for extraction and LVS purposes. References to these cells will disappear from the **Dump Phys Netlist** listing, unless the boolean variable PnetListAll is set, in which case they are listed. Flattening of electrical cells is similar. The nets, devices, and subcircuits of the flattened instance master are linked into the containing schematic. The containing schematic does not display this visually, this affects only the internal data structures.

Logical flattening can be controlled by the user in two ways: with the flatten property, and with the FlattenPrefix variable. The variable can be set from the **Cell flattening name keys** group in the **Net and Cell Config** page of the **Extraction Setup** panel, from the **Setup** button in the **Extract Menu**.

Of these two methods, use of the property is most efficient and flexible, and the setting is inherently persistent as the property value is saved in the layout file. It is the only means by which the flattening of individual instances can be controlled.

The flatten property can be applied to electrical and physical cells and cell instances. It can be applied to the current cell with the **Cell Property Editor** available in the **Edit Menu**. The **Add** menu in this panel provides a **flatten** choice in both electrical and physical modes. When the flatten property is applied to a master cell, instances of the cell will by default always be flattened in the extraction system. However, flatten properties can be applied to cell instances as well, using the **Property Editor** from the **Edit Menu**. Again, this is true in both electrical and physical modes. If an instance is given a flatten property, and its master also has a **flatten** property, the instance will **not** be flattened. If an instance has the property and the master does not, then that instance, only, will be flattened. Thus, when applied to a cell instance, the flatten property inverts the flattening status implied by the master, for that instance.

In earlier *Xic* releases, logical flattening was controlled with the FlattenPrefix variable. The variable can be set to a list of pattern-matching tokens which match the names of cells to be flattened. Cell names can be matched by prefix, suffix, or verbatim. The user is required to set this variable before extraction, a step that can be avoided by use of the flatten property instead. Unlike the flatten property, the FlattenPrefix variable applies to physical (layout) cells only. The **Cell flattening name keys** text entry area in the **Net and Cell Config** page of the **Extracting Setup** panel from the **Extract Menu** can be used to set the FlattenPrefix variable. The variable can also be set from a startup file or the technology file. See the description of the variable for the property string syntax.

16.5 Extraction System: Group/Net Naming

It is possible to apply a name to a physical wire net (or group) by use of special labels. The group name will be used in output when appropriate. It will also be used in association to match to electrical nets which have the same name (see 7.11).

Net name labels are created in the same way as any other label in physical mode. There are two requirements that must be met for the label to be applied.

1. The label origin mark must reside over or touch an object on a layer with the `Conductor` attribute. This is generally applied in the layer blocks of the technology file to metal or otherwise conducting layers.

2. The layer name of the layer-purpose pair on which the label resides must match that of the metal object above. The purpose name must match the special purpose name defined for this purpose within *Xic*. By default, this purpose is named "`pin`".

For example, suppose we have a box on a metal layer `M1`, which has the default "`drawing`" purpose. To name the conductor group containing this box, we create a label, containing the name, on the layer-purpose "`M1:pin`", and place this so that the label origin mark, which is marked with a tiny ghost-drawn cross during placement, will be located in or touching the box.

The assumed "`pin`" purpose name can be changed with the PinPurpose variable, or equivalently with the **Net label purpose name** text input area of the **Net and Cell Config** page of the **Extraction Setup** panel from the **Extract menu**.

Alternatively, the PinLayer variable can be set to a layer name, and all netname labels must reside on this layer. In this case, the pin purpose is not recognized. This is for compatibility with older libraries and is not recommended.

Physical groups can acquire a name in the following ways, listed in priority order.

1. An element of the group contains a net name label. Only one name is allowed, all net labels are ignored if conflicting net names are found. The net may have arbitrarily many name labels, but each must specify the same name logically. Names are by default case-insensitive, and different subscripting delimiters are taken as equivalent. Net name labels override any other name that might be associated with the group.

2. If an otherwise unnamed group contains a scalar cell terminal after association, the group name will take the terminal name. There can be at most one terminal, if more than one the terminal names are ignored.

3. After association, otherwise unnamed groups will take any assigned name of the corresponding wire net. Net names can be applied in electrical mode with the **Node (Net) Name Mapping** panel from the **nodmp** button in the electrical side menu. Electrical nets can also be named with wire labels, or with the named terminal devices provided in the device library.

During association, electrical nets and physical conductor groups with the same logical name will be associated without further testing. At this point, the only names that may apply to the conductor group are from name labels, or from cell terminals that have been placed into the layout by the user and therefor have the FIXED flag set. Thus, liberal but **correct** use of net name labels can speed up the association operation.

There is a provision for automatically generating or updating net name labels after association, with the **Update net name labels after association** check box in the **Net and Cell Config** page of the **Extraction Setup** panel, which is displayed with the **Setup** button in the **Extract Menu**. This tracks the state of the UpdateNetLabels variable, which can be set or cleared directly to set this mode.

This mode should be used only when layout of a cell is complete and LVS passes. Adding additional net name labels by forcing association with this mode active before saving the cell to a library can maximize association efficiency wen the cell is used in the future.

It is also possible to ignore net name labels entirely by use of the **Ignore net name labels** check box in the same panel, or equivalently by setting the IgnoreNetNameLabels variable. It is unlikely that a user will require this with any frequency.

The pre-association net names, as obtained from net name labels and user-placed cell terminals, can explicitly imply connectivity in LVS, thus accounting for "split nets". A split net is a logical conductor group that consists of two or more physically disconnected conductor groups. For example, the cell schematic may show a simple wire distributing power to all parts of a cell. The physical implementation, though, might consist of a power ring in the parent or another cell, that runs over and makes contact at various points in the cell. Without the power ring, the cell contains multiple locations that should be, but are not, connected together. Association and LVS will fail in this case, however there is a way to detect and accept this type of case.

If each piece of a split net has a separate and of course logically equivalent net name, then setting the **Merge groups with matching net names** check box in the **Net and Cell Config** page of the **Extraction Setup** panel from the **Extract Menu** will cause association to logically merge these groups, allowing LVS to pass. Equivalently, the MergeMatchingNamed variable, which tracks the check box, can be set or cleared to the same effect.

Although it is required that this mode be active for successful LVS when the top-level cell contains split nets, it is not required otherwise, even if cells lower in the hierarchy contain split nets.

16.6 Extraction System: Ground Plane Handling

The extraction algorithm can handle the situation where there is a single ground plane layer, either clear or dark field. Groups connected to ground are always assigned to group number zero. Group zero is only used when a layer has been identified as a ground plane through one of the keywords.

By default, handling of a GroundPlane (clear field) layer is the same as for other Conductor layers, however, in the top-level cell, the largest area group extracted on this layer is assigned to group 0, the ground group. There an alternative mode where all areas of the layer, in any cell, are assigned to the ground group.

There are two levels of support for a dark-field ground plane, indicated by the presence or absence of the MultiNet keyword following "`GroundPlaneClear`". The simplest situation is where the MultiNet keyword is absent. In this case, terminals and contacts with no connection, which would otherwise connect to the GroundPlaneClear layer if that layer were present, are assigned to group 0 (ground).

For example, suppose the technology file contained the following lines:

```
Layer M0
GroundPlaneClear
...
Layer I0
Via M1 M0
```

In this case, an area of `I0` over an area of `M1` and *not* over an area of `M0` would indicate a connection of the `M1` area to ground.

To repeat, if the MultiNet keyword does not appear, then all areas *outside* of the GroundPlaneClear layer geometry are assumed to be above ground. Vias and Contacts that have been specified for the ground plane layer will make contact to ground in the *absence* of the ground plane layer.

Although this sometimes works for simple cells, it can lead to trouble. Suppose that an island of ground plane metal is used as part of the metalization for the chip pads. This would appear as a hole in the displayed representation of the ground plane layer. Then each pad will be extracted as shorted to ground!

If the MultiNet keyword is given following the GroundPlaneClear keyword, then an internal layer, which is the inverse polarity of the ground plane layer, will be created and used for extraction purposes. The algorithm used for inversion can be specified by an integer 0–2 which optionally follows "`MultiNet`". There are also **!set** variables which parallel the technology file keywords. Complete information can be found in 16.8.

16.7 Extraction System: Measurement Caching

During association, and when a physical netlist is being created, measurements may be performed on devices in the layout to extract parameter values associated with the device. This may, for example, be the resistance of a resistor device, or geometrical factors associated with a MOS transistor device.

The measurement may be rather compute intensive and time consuming, thus *Xic* supports a means for caching measurement results. The caching can radically reduce the time required to associate the circuit, but it requires that the user intervene to update the cached values if the underlying geometry changes. This **does not** happen automatically. Thus, measurement caching is disabled by default. The caching is enabled by setting the UseMeasurePrpty variable, or equivalent checking the **Use Measurement results cache property** check box in the **Device Config** page of the **Extraction Setup** panel from the **Extract Menu**.

Every device can have a "data box". This is created automatically in *Xic* on a layer-purpose pair named "`device:xicdata`". The box coordinates are set to the extracted body bounding box of the device.

The measurement results are saved to a measures property that is applied to the data box. This is property number 7106. The property string is set to a space-separated list of numbers, or colon-separated pairs of numbers, representing measurement results, or non-permuted and permuted measurement results if the device has permutable contacts and the measurement result changes on permutation. The ordering is the same as the order of measurement requests in the device definition block.

When a cell is read, by default the data box and measures property, if present, are ignored. If the UseMeasurePrpty variable is set, and the NoReadMeasurePrpty variable is not set, the values from the measures property will be used when parameters are needed, and no values will be computed if the measures property is found. The NoReadMeasurePrpty variable tracks the state of the **Don't read measurement results from property** check box in the **Device Config** page of the **Extraction Setup** panel from the **Extract Menu**.

After association, if the UseMeasurePrpty variable is set, the data boxes and measures properties will be created if necessary, and updated with the current measurement values.

Thus, to globally update the cached measurement values, one can use the following procedure.

1. Press the **Clear Extraction** button in the **Extraction Setup** panel. This will invalidate the present extraction state.
2. Make sure that the **Use measurement results cache property** check box in the **Device Config** page of the **Extraction Setup** panel is checked.
3. Make sure that the **Don't read measurement results from property** check box on the same page is also checked.
4. Press the **Do Extraction** button in the same panel. This will run grouping, extraction and association operations. Measured values will be computed, since any cached values are ignored due to NoReadMeasurePrpty being set. After association, the measures properties are updated to the newly computed values.
5. Un-check the **Don't read measurement results from property** check box. Save the cell hierarchy.

16.8 Extraction System: Setup and Configuration

Use of the extraction features requires setting certain keywords and data blocks in the technology file. There are three types of entries:

1. Keyword descriptions in the various physical layer blocks, including the extraction (see A.6.4) and physical property keywords see A.6.5). These define the layers as conductors and insulators. See the references for complete information.
2. Global attribute variables (see E.24). There are a few such variables for the extraction system, which provide such things as the substrate dielectric constant.

3. The device blocks (see 16.8.1), which appear after the layer blocks in the technology file. These define device structures to be recognized and extracted.

In addition, the user may wish to further customize the technology file by adding scripts which perform some extraction-related function, such as generating temporary layers.

If the user wishes to define a customized format for physical or electrical netlist output, an entry in the format library file can be added. The format library contains scripts which provide formatting for the commands in the **Extract Menu** that produce netlists. Section 16.8.3 provides more information on this capability.

16.8.1 Device Blocks

Physical characteristics of devices which are candidates for extraction are specified in device blocks in the technology file. The device blocks are located in the technology file after the physical layer definitions. These specifications enable automated extraction of circuits from physical layouts.

Devices are specified in the technology file through a block of lines keyed by the word "`Device`" and ending with "`End`". An example is below:

```
Device
Name res
Prefix R_
Body R2
Contact + M2 I1B&R2
Contact - M2 I1B&R2 ...
Permute + -
Depth 1
Merge S
Measure Resistance Resistance
LVS Resistance
Spice %n% %c%+ %c%- %ms3%Resistance
Cmput Resistor %e%, resistance = %ms3%Resistance
Value %m%Resistance
End
```

There can be no text following `Device` in that line. The block must terminate with `End`.

The device block in the example specifies a resistor device:

- The resistor body consists of areas of layer `R2`.
- Contact is made to conductor `M2` through a region of `I1B` (which represents a physical via).
- The resistor can have arbitrarily many contacts (`...` given in second `Contact` line). This will be decomposed into a network of two-terminal resistors by the extraction system.
- The (two) terminals are interchangeable (`Permute` given).
- Parts of the resistor can be found in subcells (`Depth` of 1).
- Two-terminal resistors in series and in parallel will be merged iteratively into simplified networks.

- The resistance of the structure will be measured and reported.

The keywords are described in detail below.

`Template` *template_name argument ...*
This will access the device template with the given *template_name*. Text from the template will be inserted into the current device block, after macro, variable, and argument substitution. For argument substitution, forms like "`$(_`*N*`)`", with N being a positive integer, will be replaced by the N'th argument, with any quote marks stripped. If an argument contains white space or other strange characters, it should be double-quoted.

The template can provide all of the keyword text for the current device block, or additional keywords from the list below can be provided to supplement the template. It may not be possible to redefine an alredy defined keyword however.

Name *device_name*
The *device_name* names the device, which should match a device cell name. The name can be that of a parameterized cell, or a regular device cell from the device library (`device.lib`) file, or a device cell from some other source. This line is mandatory.

Two or more device blocks can use the same name if they have different Prefix entries. This might be useful, for example, if there are two resistor layers in a process. A device block would be needed to describe resistors on each layer.

Prefix *prefix*
This is a prefix that is prepended when formulating the name for the device used in output. This is optional, as a prefix can also be defined in the output formatting. The first letter of the prefix should match the expectations of the SPICE simulator or other tools to be used. If two or more device definition blocks have the same Name field, they must have different Prefix fields. Further, each definition must have identical contact and bulk contact names, and order, and identical permutable contact names.

Body *expression*
The Body keyword specifies the "core" physical feature of the device. The specification is a layer expression. Each individual region where the expression is true defines a potential instance of the device. This keyword is mandatory.

Contact *name layer expression* [...]
For each contact of the device, there should be a Contact line. The first token following Contact is a name for the contact, which should match the corresponding name used in the node property of the device in the device library file. The second token is a layer name of a conductor layer which is used to contact the device. The remainder is an expression which identifies the contact area. The contact is identified as a region inside the device bounding box where the expression is true. Multiple contacts using the same expression can be given, and each will select a different region. The device bounding box is the bounding box of the body area, after the Bloat operation (see below).

If the Contact line ends with "`...`" (three periods) there can be more than one of that type of contact. Ordinarily, there is a one-to-one correspondence between contacts specified and contacts in the device instance. With the ellipses, device instances will include as many of that type of contact as can be found. Thus, such devices no longer have a fixed number of contacts. The ellipses can *not* appear in the first Contact line, but may appear in the second and/or subsequent Contact lines.

The ellipses feature presently supports multi-contact resistors. A multi-contact resistor is replaced internally by a network of two-terminal resistors, which are used in the netlist output. To enable multi-contact resistor support, the second contact specification in the resistor device block should end with "`...`", for example

```
Contact + M2 I1B&R2
Contact - M2 I1B&R2 ...
```

This specifies that as many of the second type of contact as can be found will be extracted. Without the "`...`" only two contacts would be extracted. The ellipses can not occur on the first contact line, but may occur on other lines, and may occur more than once, though no standard devices use this feature presently. In general, this implements device extraction with arbitrary numbers of certain contacts.

Internally, a conductivity matrix is computed from the body and contact geometry, and this is used to compute the effective values of the two-terminal resistors that are used to implement the multi-contact resistor. Resistors that would have very high values (larger than 100 times the smallest value) are not added, so that linear multi-contact resistors decompose as one would expect.

The decomposition occurs before the serial/parallel merging, so that the components of the decomposition are candidates for merging, if merging is enabled (see the Merge keyword below).

`BulkContact` *tname level* [*name* | *bloat layername expression*]

This is a special form of a contact specification that applies to well and substrate connections, which may be treated differently than other contacts. The *tname* is the contact name. This is followed by an integer *level* which determines how the contact is handled during extraction. Possible *level* values are as follows. The remaining entries in the line depend on the *level*. There can be at most one `BulkContact` line in the device description.

0

Check in cell during device extraction. This requires that a *bloat* value, *layername*, and *expression* follow the level number. The body bounding box is (logically) bloated by the *bloat* value given (in microns). Within the bloated area, a region where the *expression* (a layer expression) is dark, that is connected to the conductor *layername* must exist. If there are multiple areas, the one closest to the bounding box center is taken as the contact area. If there is no such area, the device will not be recognized. The search hierarchy depth for the contact is to all levels.

1

Ignore this in extraction. The level value must be followed by a *name*, which is the name of a global net. The contact will be assumed to connect to that global net. Use this mode only if it is absolutely certain that physically the bulk contact is connected properly, as this mode does no checking. Use this at your own risk.

2

Check deferred. This requires that a *bloat* value, *layername*, and *expression* follow the level number. During extraction and association, if the contact can't be resolved within its containing cell, *level*=2 contacts are ignored as for *level*=1.

During LVS, a special "stamping" test is run over the complete hierarchy, and any errors found are reported with the top-level cell. This test searches for unresloved *level*=2 contacts in the hierarchy. It will try and resolve the contact at the top level (thus taking account of all geometry in the hierarchy). If unsuccessful, the device location as reflected to the top level coordinates will be listed in the stamping report, and LVS will not succeed.

Bloat *increment*

This will expand the body bounding box by *increment* (in microns) for the purpose of identifying contacts. For example, a MOS transistor body is the intersection of CAA and CPG. The source and drain contacts can be specified as the regions of the body bounding box after a bloat that cover CAA but not CPG.

ContactsOverlap

Ordinarily, device contact areas can not overlap. The extracted contact areas are clipped against one another to enforce this. Giving this keyword allows overlap, which is necessary for some vertical device structures.

Permute *name1 name2*

The names are the names of contacts that can be permuted to enable association when comparing to a schematic. This applies to devices such as resistors, and to the source and drain of MOS devices, or to any device containing a contact pair that are geometrically identical to the extractor. There must be exactly two names following "Permute", and only one Permute line is allowed.

Depth *depth*

The *depth* is the hierarchy depth extracted for the device, default is 0, meaning all device structure should appear is the current cell. The value can be an integer, or 'a' to look at the full hierarchy.

Find [*device_name*][*.prefix*]

This will cause a device with the given name and prefix to be searched for in the current device's bounding area, and added to an internal list. Any number of Find directives can be applied. If two or more directives look for the same name/prefix, they will return different instances. Currently, this is used only for identifying inductors in a mutual inductor device.

Merge [*arg*]

This optional keyword specifies how to handle parallel and series connected instances of the device for parameter extraction. There is an optional argument. Merging implies that multiple devices are combined internally and reported as single devices in netlists and SPICE output. If both parallel and series merging are enabled, the merging process is iterative, and will continue until no further merging is possible.

If no Merge keyword appears in the device block, no merging is done for that device. Only the first two characters of the *arg* are tested, case insensitively, and any remaining characters are ignored. Series merging will be enabled only for two-terminal devices that have the Permute keyword applied, i.e., typically resistors, capacitors, inductors.

arg	**Merge**
no *arg*	parallel
`"s"`	parallel and series
`"ns"` or `"sn"`	series
unrecognized	error

Merging can also be controlled by the variables NoMergeParallel and NoMergeSeries which are booleans which can be set with the **!set** command. The variables suppress merging of the indicated kind, parallel or series, for all devices.

Merging can also be suppressed on an individual device basis by applying a NoMerge property to an object that is used in the body of the device. This property can be added with the **Property Editor**.

Merging can lead to confusion, particularly when users are experimenting. Unless the aggregate has external connections, it is likely to be merged down to a single device in ways which may be surprising.

Example:
The **Show computed parameters of selected device** option of the **Enable Select** command mode in the **Show/Select Devices** panel from the **Device Selection** button in the **Extract Menu** is useful for displaying the values of extracted devices, and shows the effect of merging. When resistor networks are merged, *Xic* will merge series resistors if there are no other connections at the common node. Sometimes, this will lead to a configuration that is not intended or desired, for example if the desired end terminal of the network is connected to two resistors only, that node might be merged away. *Xic* will merge devices arbitrarily if there is insufficient information available to uniquely define how merging is to be done.

One way to prevent this from happening is to use temporary virtual terminals:

1. Switch to electrical mode.
2. Enter the **subct** side menu command.
3. Press **Ctrl** and click anywhere in the drawing window. A terminal marker will appear. Dismiss the **Terminal Edit** pop-up, and switch back to physical mode.
4. Press the **Setup** button in the **Extract Menu** to obtain the **Extraction Setup** panel. Press **Edit Terminals** in the panel. A terminal mark should appear to the lower left of the bounding box of the current cell.
5. Move the terminal mark to the desired network end terminal metal.

This node now has a (phony) terminal, so it won't be merged. Don't forget to go back and delete the terminal when done.

The way the parameters are computed upon merging is determined by the Measure keyword (see below). Series merging is applicable to resistor, capacitor, and inductor-type devices.

Measure Keyword	**Parallel Action**	**Series Action**
BodyArea	Sum	Sum
BodyPerim	Sum	Sum
BodyMinDimen	Min[1]	Min
CArea	Sum[2]	-
CPerim	Sum[2]	-
CWidth	-	-
CNWidth	-	-
CBWidth	Average	Sum
CBNWidth	Sum	Average
Resistance	Parallel Resistance	Sum
Inductance	Parallel Resistance	Sum
Mutual_Inductance	not implemented	not implemented
Capacitance	Sum	Parallel Resistance

Notes:
1) Although the minimum of the multiple sections is used, for MOS devices each value is typically the same.
2) If devices of the same type share a contact, the contact area and perimeter are divided equally between the devices.
The fields with '-' above are invalid, and return 0 if accessed.

The "`Merge M`" feature of earlier releases is no longer supported. This would average the parameters of parallel-connected MOS devices, and automatically add the "`M=`" (multiplier) parameter to the SPICE output line. Now, the merging behavior is as described above, and no multiplier is automatically added to the SPICE line. The Sections measurement keyword (below) can be used to explicitly format the SPICE output to use the multiplier parameter, if desired.

Measure *mname expression* [*precision*]

The Measure keyword allows geometrical information to be extracted from the device, which is listed with the **Dump Phys Netlist** command in the **Extract Menu** and used in other commands in the **Extract menu**.

mname

A name for the parameter te be extracted. This is arbitrary but should be unique for the device. This is the name by which the particular measurement result is referenced.

precision

The optional *precision* is a non-negative integer which applies to comparing electrical and physical values in layout vs. schematic (LVS) testing. The default is 2 if not given. If the value given is n, then the two values must agree to a part in 10^n, e.g., to within 1 percent for the default value of 2.

expression

The *expression* consists of an expression in the format recognized in scripts, where the variables are either the names from previously defined Measure lines (in the current device block) or the keywords below. The expression is evaluated during extraction yielding the result of the measurement. The math functions are available in the *expression* as are all of the math operators. There can be arbitrarily many Measure lines.

Below is a list of the "primitive" measurement tokens which can appear in the measurement expressions. In several cases, the token consists of three fields, separated by '.'. The additional fields supply modifiers to the primitive measurement indicated by the token name (the first field).

The basic unit of length is one micron.

Sections

This returns the number of components of the device, which will be greater than one if the device is an aggregate of several series or parallel-connected devices (Merge enabled).

BodyArea

The area of the region where the Body expression is true.

BodyPerim

The perimeter length where the Body expression is true.

BodyMinDimen

This is a minimum dimension computed using the body geometry. There are several ways that this can be computed, depending on other keywords and the device type. The default algorithm first decomposes the body shape into a trapezoid list. the mid-height width and height of each trapezoid is added to a histogram, weighted by the other value. For example, width=2, height=3 would add to the histogram 2 with weight 3, and 3 with weight 2. When done, the value with the largest weight will be taken as the BodyMinDimen. If a tie, the smaller value is used. This is effective on structures where a "line width" is an applicable concept.

However, if the SimpleMinDimen keyword is found, the BodyMinDimen will instead be the smallest width or height found. This was the default algorithm in releases prior to 3.1.6. The result of the simple algorithm is less useful, as, for example for a serpentine structure, it could be the line width, or the spacing.

There is yet another BodyMinDimen algorithm, associated with MOS devices and indicated by the ContactMinDimen keyword. With this keyword, and if a Permutes contact list is given, the BodyMinWidth will be the distance between the inside edges of the two contacts in the Permutes list. This is the default for recognized MOS devices, i.e., devices whose prefixes start with 'm' or 'M', and overrides SimpleMinDimen if both are applicable.

For MOS devices, where it is assumed that the gate length (source to drain) is constant, i.e., the gate is a strip that can meander arbitrarily, even forming a loop, for device size measurements, one can specify

$length =$ BodyMinDimen
$width =$ BodyArea/BodyMinDimen

The next four keywords have two optional trailing fields. The value in each field must be a single digit. The digit corresponds to a Contact, in order of appearance in the device block, starting with 0. If one or both fields is left off, the effective entry is 0. If both contacts are given the same digit, the second one is incremented. Thus, leaving off the trailing fields is equivalent to ".0.1". If the indices don't point to an existing contact, or are not single digits, the measurement will fail. These are illustrated in Figure 16.1.

The "body bounding box" is the rectangular region encompassing the Body objects, before any bloat.

CWidth[.*n1*.*n2*]
: The width of the first contact, along a line connecting the first contact with the second contact.

CNWidth[.*n1*.*n2*]
: The width of the first contact, normal to the line connecting the first contact to the second contact, measured at the contact bounding box midpoint.

CBWidth[.*n1*.*n2*]
: The width between the first contact and the second contact, which lies over the body bounding box.

CBNWidth[.*n1*.*n2*]
: The length of the line normal to the line between the first contact and the second contact, over the body bounding box, passing through the center of the body bounding box.

Example:

```
Contact s CAA CAA & !CPG
Contact d CAA CAA & !CPG
Contact g CPG CAA & CPG
Measure Length CBWidth.0.1 * 1e-6
Measure Width CBNWidth.0.1 * 1e-6
```

Note that the conversion to meters is included in the Measure lines in the example above.

The following two keywords contain two trailing fields, which are mandatory. The first field contains a contact index as above. The second field contains the name of a layer.

CArea.*n1*.*lname*
: Construct a single polygon from the connected objects on the named layer, one of which intersects the bounding box of the given contact. The area of the polygon is returned. Note that the constructed polygon can extend outside of the device's bounding box. If the device being measured is merged, then the result is the sum of the results from each component.

CPerim.*n1*.*lname*
: Measure the perimeter length of the polygon constructed as above. If the device being measured is merged, then the result is the sum of the results from each component.

When measuring with CArea/CPerim (for MOS ad/as/pd/ps), there is a test to see whether the area intersects other device contacts from the same device type. If a contact is shared between two devices, e.g., common active layer for two series-connected MOS devices, the following algorithm is invoked.

For the shared contact:

1. Compute the total contact area and perimeter for both devices.
2. Compute the area and perimeter for the contact area common to both devices.
3. Subtract 1/2 this value from the parameters computed in the first step.

This algorithm should work whether or not the devices are multi-component and merging is enabled.

Example:

```
Contact s CAA CAA & !CPG
Contact d CAA CAA & !CPG
Contact g CPG CAA & CPG
Measure AS CArea.0.CAA
Measure AD CArea.1.CAA
Measure PS CPerim.0.CAA
Measure PD CPerim.1.CAA
```

Resistance
Extract the resistance value (see below).

Capacitance
Extract the capacitance value. The returned capacitance value is given by the BodyArea times the capacitance per unit area, plus the BodyPerim times the capacitance per unit length. The capacitance values are specified in a Capacitance line in the layer block of one of the layers defining the device body, i.e., the layers mentioned in the Body line. If no body layer contains a Capacitance specification, or if both parameters are zero, an error results.

Inductance
Extract the inductance value (see below).

Mutual_Inductance
Extract the mutual inductance value. This is not yet implemented.

The internal device definition structure contains a flag that if set causes the device to be treated as a MOS transistor. There are a few MOS-specific tests and operations found in the extraction system, which are enabled by the flag. By default, the flag is set if the device `Prefix` starts with 'm' or 'M'.

There are also flags that are set if the device is determined to be n-type or p-type. Presently, we only set these for MOS devices. By default, if the device name begins with 'p' or 'P', the device is assumed to be p-type, otherwise it is taken as n-type.

NotMOS
If given, the flag that indicates that the device is a MOS transistor will not be set, as it would normally be if the `Prefix` starts with 'm' or 'M'.

MOS
If this keyword is given, the flag indicating that the device is a MOS transistor will be set. This overrides NotMOS.

NMOS
Flags will be set to indicate that the device is an n-type MOS transistor.

PMOS
Flags will be set to indicate that the device is a p-type MOS transistor.

Ntype
Flags will be set to indicate that the device is n-type. This is meaningful only for MOS transistors at present.

Ptype

Flags will be set to indicate that the device is p-type. This is meaningful only for MOS transistors at present.

SimpleMinDimen

When given, and the ContactMinDimen is not applied or not applicable, the BodyMinDimen measurement will be the smallest trapezoid width or height found in the decomposition of the body shape. This was the default algorithm in releases prior to 3.1.6, but a better algorithm is the new default.

ContactMinDimen [`y/n`]

This keyword has a dual purpose: to impose a MOS-like BodyMinDimen computation on other device types, and to turn off the use of this algorithm in MOS devices, which use this algorithm by default.

Recognized MOS devices are devices that have the internal flag set as mentioned above. The device must also have a `Permutes` list for this algorithm to apply. Most MOS devices have permutable source and drain contacts.

In recognized MOS devices with permutes, the default BodyMinDimen calculation is to set this to the distance between the inside edges of the two contacts listed in the Permutes list. Thus, the BodyMinDimen will always be the device length (source/drain spacing) even if the source-drain spacing is larger than the device width. The simple "line width" algorithm normally applied for the BodyMinDimen would be ambiguous as to whether the BodyMinDimen is the device length or width.

If ContactMinDimen n is given for a recognized MOS device, the line width algorithm will be used. The "`n`" can actually be one of many tokens that indicate negativity, such as "`no`", "`false`", "`off`", "0", etc., case insensitive, but the token must appear.

For devices that are not recognized MOS devices, the line width BodyMinDimen algorithm is used by default. However, if ContactMinDimen y is given, and the device has a Permutes list, the BodyMinDimen will be computed as for MOS devices. The "`y`" can actually be missing, or can be one of many possible tokens that indicate truth, such as "`yes`", "`true`", "`on`", "`1`", etc., case insensitive.

LVS *measure_expr* [*spice_name*]

This keyword instructs *Xic* to perform a parameter comparison as part of LVS. The *measure_expr* is either one of the names used for a Measure statement in the device block, or a single-quoted expression involving constants and names from Measure statements. The *spice_name*, if given, is the token used in SPICE element lines to designate the parameter, e.g., "l", "w", "area". This can be blank if comparing to an element value which is given as a leading number, i.e., resistance, capacitance, etc. The LVS directives must appear after the referenced Measure line.

Examples:

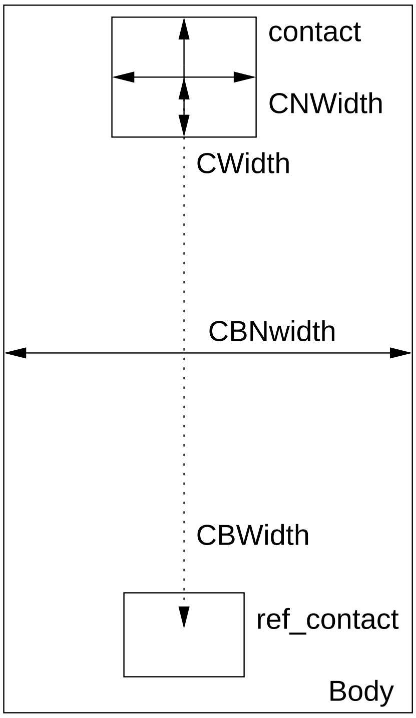
```
Measure Area BodyArea*1e-12
LVS Area area

Measure Resistance Resistance
LVS Resistance
```

Any number of LVS lines can appear in a device block.

Spice *specification_args*

The Spice keyword specifies the format for the SPICE output part of the listing from the **Dump**

Figure 16.1: The distances returned by the various width measurement keywords to the device block Measure line.



Phys Netlist command in the **Extract Menu**. The specification is copied verbatim, except for the following substitutions:

\n
: The character sequence '\n' is replaced by a newline in the expanded text. Note that the next token should probably begin with the SPICE continuation character '+' for the SPICE output to be interpreted correctly.

\t
: The character sequence '\t' is replaced by a tab character.

%c%*cname*
: The *cname* is a contact name (from a Contact line). This token is replaced with the group number of the contact.

%m[g | s[*N*] | f[*N*] | e[*N*]]%*mname*
: The *mname* is a name from a Measure line. The token is replaced with the result of that measurement. One of the characters s, g, f, e can follow the 'm'. The s, f, e can be followed by an optional digit. These select the format of the printed result.

  g
  : Use the "best" numeric format (the default if no modifier given).

  s*N*
  : Use SPICE abbreviations, with *N* decimal places.

  f*N*
  : Use fixed point notation, with *N* decimal places.

  e*N*
  : Use exponential notation, with *N* decimal places.

  If *N* is not given, the default is 5 digits.

Above, *cname* and *mname* can be followed directly by '%' and other text, for a concatenation function. For example "`L=%m%Length%u`" might be replaced with "`L=.8u`".

%n%
: This token is replaced with a name for the device, which consists of the Prefix (if given) followed by an index count for the device type.

%p *lname pnum*%
: This token is replaced by the text of a physical property. The *lname* is the name of a layer, and space after the 'p' is optional. The *pnum* is a non-negative integer. Each of the objects on *lname* that intersect the device bounding box is checked for a property with number *pnum*. The string of the first such property found is used. This enables property text to appear in device output, in particular it provides a means to coerce a value or other parameter.

%e%
: If the electrical dual of the physical device is known, the %e% is replaced by the name of the electrical device. If no dual is known, the behavior is the same as %n%.

%f%
: The substitution %f% is equivalent to %e% except that if the dual device is unknown, the token is simply ignored.

Each of the substitution tokens can take an optional integer after the first %, which indicates that the token refers to the device in the n'th Find line (0 is the same as no integer).

Example:

```
Device
Name mut
Prefix K
...
Find ind
Find ind
Spice %n% %1n% %2n% ...
```

The Spice line prints the name of the mut device, followed by the names of the two inductors.

%model%
: Replaced by contents of the Model line (see below).

%value%
: Replaced by contents of the Value line (see below).

%param% or %initc%
: Replaced by contents of the Param line (see below).

The Spice line is used in **Dump Phys Netlist** output and internally by **Source Physical** command.

Cmput *specification_args*
: This specifies the format used in printing the device parameters from the **Show computed parameters of selected device** option of the **Enable Select** command mode in the **Show/Select Devices** panel The substitutions are exactly as those of the Spice keyword. For example:

```
Device
Name res
...
Measure Resistance Resistance
Cmput Resistance = %m%Resistance ohms
End
```

Model *specification_args*
Value *specification_args*
Param *specification_args*
: These keywords specify a format string to use when creating "property strings" from the extracted parameters of a physical device, to be used for comparison or updating the properties of the corresponding electrical device. These are used in the **Source Physical** command, and in the **Show elec/phys comparison of selected devices** option of the **Enable Select** mode in the **Show/Select Devices** panel. This panel is brought up by the **Device Selections** button in the **Extract Menu**.

The format is the same as is described for the Spice line, however the escapes `%model%`, `%value%`, and `%param%` are not recognized.

The Model, Value, and Param lines are used internally when comparing physical devices to their electrical counterparts. This is done, for example, in the **Show elec/phys comparison of selected devices** option of the **Enable Select** mode in the **Show/Select Devices** panel. This panel is brought up by the **Device Selections** button in the **Extract Menu**.

The Set *varname* = *something* construct in the technology file can apply to lines in device blocks, however the Set keyword must appear outside and before the block. Device block lines can contain `$(`*varname*`)` tokens, which are replaced with *something* before the line is parsed.

The format specifications for the Spice, Cmput, etc. lines can contain the eval(...) construct. The argument to eval is evaluated as a mathematical expression, and the result replaces the entire construct. Unlike elsewhere in the technology file, in these lines this construct is evaluated when the line is used, and not when the technology file is read.

The extraction mechanism can be tested with the **!find** command, or with the device listing capability in the **Show/Select Devices** panel from the **Device Selections** button in the **Extract Menu**.

Xic contains functionality for accurately calculating resistor values of arbitrarily shaped resistors. Resistance extraction is accomplished by dividing the resistor logically into a regular grid. The center of each grid is a "node" that is connected by resistance to adjacent nodes. Thus, the problem becomes one of solving a large lumped resistor mesh.

Best accuracy is obtained when the grid falls on all the resistor and contact boundaries. It is not possible to find such a grid in general, however if a layout grid is used and all corners are on-grid, and all edges are Manhattan, then tiling will be possible. It may be the case that tiling is possible, but the tile is so small that the computation time is unacceptable.

For structures that can't be tiled efficiently, a set of edge-dependent heuristics is used to modify the matrix elements to account for the local area deficit or surplus.

There are four variables that can be used to configure the extractor. The default values lean toward speed over accuracy. By default, tiling is not attempted, and the grid spacing will be selected so that each resistor contains 1000 grid cells.

RLSolverDelta

Value: floating point >= 0.01.
It this value is set, the resistance/inductance extractor will assume this grid spacing, in microns. The number of grid cells enclosed in the device will increase for physically larger devices, so that larger devices will take longer to extract. If this variable is set, the other RLSolver variables are ignored. Setting this variable may be appropriate if all resistors are "small" and dimensions conform to a layout grid.

RLSolverTryTile

Value: boolean.
If set, the extractor will attempt to use a grid that will fall on every edge of the device body and contacts. The device and contact areas must be Manhattan for this to work. If such a grid can be found, and the number of grid cells is a reasonable number, this will give the most accurate result.

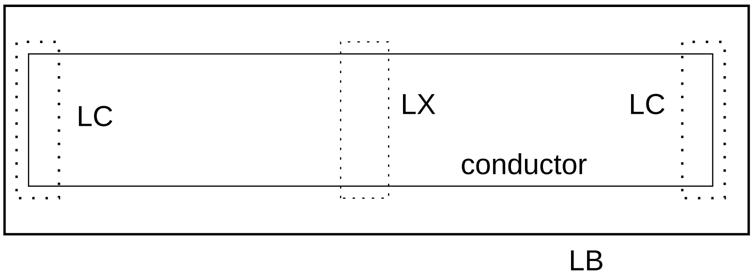
RLSolverGridPoints

Value: integer 10–100000.
When not tiling (RLSolverTryTile is not set), this sets the number of grid points used for resistance/inductance extraction. This number will be the same for all device structures, so that computation time per device is nearly constant. Higher numbers give better accuracy but take longer. The value used if not set is 1000.

RLSolverMaxPoints

Value: integer 1000–100000.
When tiling (RLSolverTryTile is set), the maximum number of grid cells is limited to this value. If the tile is too small, it will be increased in size to keep the count below this value, in which case the tiling will not have succeeded so there may be a small loss of accuracy. Using a large number of grid points can take a long time. The value used if not set is 50,000.

Figure 16.2: Illustration of the configuration of layers LB, LC, and LX for extracting inductance from a conducting strip. The LX ensures terminal assignments to different groups.



The resistor solver is accessed through the device block Measure keyword "Resistance", for example:

```
Device
Name res
...
Measure value Resistance
End
```

By including the "`Measure value Resistance`", all resistances may be extracted and the values will appear in the output of the **Dump Phys Netlist** command. When computing the resistance, the layers in the Body specification are checked for an Rsh specification, or alternatively a Rho or Sigma specification along with a Thickness specification. If the resistance parameters are not found, an error results. Unlike releases prior to 3.1.6, there is no default resistance.

Xic also contains functionality to measure conductor inductance values. Inductance is extracted using an algorithm similar to resistance, i.e., square counting, but other factors are included to enhance accuracy. This assumes "microstripline" geometry, meaning a conductor separated from a ground plane by a uniform dielectric. The Measure keyword is "`Inductance`". The inductance per square is derived from the microstrip parameters for the layer, as provided with the Tline specification. A Tline specification must be given to one of the device body layers, or an error results.

Presently, the recommended way to set up inductors for extraction is through the use of three additional layers. These layers can have any name but will have the following names in this discussion:

LB
: Used to outline the inductor, will surround the region of a conductor where inductance is to be measured.

LC
: Identifies the inductor contacts (inside LB and on the conductor).

LX
: Bisects the conductor into two areas to provide separate groups for the two contacts.

These layers have been added to the `xic_tech.hyp` file provided. The "`Exclude LX`" clause must be added to the Conductor specification of the conductors to be extracted.

16.8.2 Device Templates

A device template is a device block with special syntax that allows text substitution. The `Template` line in a device block will read the referenced template while performing the substitutions. A template can be used to save common keywords associated with a class of device, for example MOS transistors, that may have several styles in a process,

The substitution text replaces forms like "`$(_N)`", where N is a positive integer. This indicates that the text of the N'th argument in the `Template` line will replace this form. Any macros or technology file variables found in the line will also be expanded at this time.

The first line of a device template consists of the keyword `DeviceTemplate`, followed by a name for the template. The template name can be just about any text word, and is used to reference the template. The final line of the template contains the keyword `End`. Intervening lines are the same as device block lines, with substitution sequences where needed. The `Device` keyword need not appear.

Device templates can bo defined in the technology file, or in a file named "`device_templates`" found in the current directory or library path. A default `device_templates` file is provided in the `startup` directory in the installation area. This contains two example templates: `NmosTemplate` and `PmosTemplate`. These provide generic recognition of MOS transistors. When a technology file is written with the **Save Tech** button in the **Attributes Menu**, only the device templates originally read from the technology file will be included. Device blocks will be written with the `Template` lines expanded.

Example:
Here's part of a device template definition.

```
DeviceTemplate mostmpl
Name $(_1)
Prefix M
Body $(_2)
...
End
```

Here's a device block in the technology file.

```
Device
Template mostmpl nmos active_layer&poly_layer
End
```

Here's the post-substitution device block.

```
Device
Name nmos
Prefix M
Body active_layer&poly_layer
...
End
```

16.8.3 Format Library File

Xic provides a mechanism for user-specified formatting of physical and electrical netlist output. Such formatting is generated by scripts found in a file named "`xic_format_lib`". This file need not exist if user formatting is not required.

An example `xic_format_lib` file is included in the distributions. This provides two examples each, for physical and electrical output. In either case, the first example is the Cadence Design Exchange Format (DEF), which is an industry-standard ASCII netlist format. The second format in each case is a simple example, not a "real" format. The example library is found in the startup directory, and can be used as-is or as a starting point for customization. The example format scripts include instructive comments.

The `xic_format_lib` file is searched for in the library search path, and the first such file found will be used.

There are three types of script that can appear in the file: those for generating netlists from physical data, those that generate netlists from electrical data, and those that format the output of LVS runs (this is not supported yet).

Blank lines, and lines that start with the '#' character, are ignored. There are four keywords (outside of the scripts) that are recognized:

`PhysFormat` *name*
`ElecFormat` *name*
`LvsFormat` *name*
`EndScript`

One of the first three of these keywords and its argument should appear on its own line ahead of a script, and "`EndScript`" should appear on its own line following a script. The *name* is the name of the format, which will appear on command or menu buttons or is given to script functions to indicate that the following script is to be used for formatting. This should be a short alpha-numeric word or phrase, and must be unique among keywords of a given type. If the *name* contains white space, it should be double-quoted.

The script lines can contain any of the script library functions and operators. All local variables are static. The script can call functions that have been previously defined in a regular library file.

When the script is executed:

- The "standard output" is to the file being generated, and not to the console window as for normal execution.
- The script will be called for each cell in the hierarchy, to a depth given in the invoking command. On each call, the "current cell" is set to the cell being processed.

When the script is executing, the following predefined variables are available for use in the script.

Name	Type Description	
`_cellname`	string	name of the cell being output
`_viewname`	string	"physical" or "electrical"
`_techname`	string	`TechnologyName` value from technology file
`_num_nets`	integer	number of wire nets in cell
`_mode`	integer	0 if physical, 1 if electrical
`_list_all`	integer	1 if **list all cells** active, 0 otherwise
`_bottom_up`	integer	1 if **list bottom-up** active, 0 otherwise
`_show_geom`	integer	1 if **include geometry** active, 0 otherwise
`_show_wire_cap`	integer	1 if **show wire cap** active, 0 otherwise
`_ignore_labels`	integer	1 if **ignore labels** active, 0 otherwise

The script will use functions that iterate through the cell and print the desired information in an order and format desired. The function library is being expanded to provide flexibility.

16.9 The Misc Config Button: Misc. Extraction Settings

The **Extraction Setup** panel appears in response to pressing the **Setup** button in the **Extract Menu**. The panel has four tabbed pages: **Views and Operations**, **Net and Cell Config**, **Device Config**, and **Misc Config**. These will be described in the sections that follow.

Common to all pages are two buttons which will invalidate or initiate extraction. This is usually done automatically — extraction is invalidated if the design changes somehow, and recomputed when needed. The buttons can force recomputation which may be useful on occasion.

Clear Extraction
Pressing this button will clear the internal validation flags, which will cause *Xic* to recompute extraction when extraction results are next needed. This is normally done automatically if the layout or schematic changes, or a setup variable is changed.

Do Extraction
Pressing this button will perform the full extraction and association, if necessary. This is normally done automatically when needed within commands. Once done, flags are set indicating the validity of the current extraction data structures, avoiding recomputation unless something changes, or **Clear Extraction** is pressed.

16.9.1 The Views and Operations Page

At the top of the page is the **Show** group, containing check boxes that make visible certain features related to extraction.

Extraction View
When the **Extraction View** check box is active, the display in the main window is based on features as known to the extraction system. Although similar to the normal display, there are important differences:

1. Only the conducting layers are shown.
2. The features from internally-flattened subcells are shown as part of the parent cell.
3. The geometry shown represents the processing from the `Conductor Exclude` directive.

4. The visible geometry includes ground-plane processing.

This viewing mode is compatible with most other commands, however when active, editing is prevented. Object and subcell selection works in the normal way, however only visible objects can be selected, which includes the "phony" objects created by the extraction system and not present in the actual geometry database.

Groups
The **Groups** check box causes the group number of each conducting object to be displayed in physical windows. This is similar to the **Nodes** check box, and is mutually exclusive with that check box.

Nodes
The **Nodes** check box causes the associated node name of each conducting object to be displayed in physical windows. This is similar to the **Groups** check box, and is mutually exclusive with that check box.

All Terminals
The **All Terminals** check box displays the terminals in the physical layout which correspond to terminals in the electrical schematic. Electrical devices and subcircuits may have terminals associated with their nodes. These terminals are used to identify the connection points in the physical database. During association, the terminals are automatically placed at the appropriate point in the physical cell.

Should association fail, unplaced terminals are grouped in an array to the lower left of the physical cell's bounding box. Also potentially visible after failure, to the right of the physical layout, are any unassigned subcircuit labels.

This check box is mutually exclusive with the **Cell Terminals Only** check box.

Cell Terminals Only
This is similar to the **All Terminals** display mode, however only the cell's connection points are shown, not the connection points of devices or subcells. This check box is mutually exclusive with the **All Terminals** check box.

The **Terminals** group provides buttons which initiate commands and modes related to terminal placement and parameters.

Reset Terms
Pressing this button will move all device and subcircuit terminals out of the layout, and array them outside the lower left of the cell. The instance connection points will become undefined. This operation cannot be undone except by re-running extraction. If the **Recursive** check box is checked, this operation will be performed recursively on all cells in the hierarchy.

The cell's contact terminals that have been explicitly placed by the user, and consequently have the `FIXED` flag set that locks the position, are not touched by this operation.

This capability is mostly for debugging. It may be entertaining to make all terminals visible, then press this button. Zooming out will reveal the terminals arrayed outside the lower left corner of the cell. Then, on pressing **Do Extraction**, the terminals will snap back to their locations within the layout. The locations of these terminals are set by *Xic*, unlike the cell connection terminals which can be placed by hand.

The same capability is available from the **!ptrms** text-mode command.

Reset Subcircuits

Pressing this button will move all of the subcircuit marker labels to an array outside of the upper right of the layout. This will undefine the subcell associations with the schematic. This operation can not be undone except by re-running extraction. If the **Recursive** check box is checked, this operation will be performed recursively on all cells in the hierarchy.

This capability is mostly for debugging. It may be entertaining to make terminals visible (which also makes the instance label marks visible), then press this button. Zooming out will reveal the instance marks arrayed outside the upper right corner of the cell. Then, on pressing **Do Extraction**, the marks will snap back to their locations within the layout. The locations of these marks are set by *Xic*, and presently they can not be manually placed.

The same capability is available from the **!ptrms** text-mode command.

Recursive

When this check box is checked, the **Reset Terms** and **Reset Subcircuits** buttons will act recursively in the hierarchy of the (physical) current cell. If not set, the operations are performed in the current cell only.

Edit Terminals

In electrical mode, this button has the same effect as the **subct** button (see 7.23) in the electrical side menu. A command is entered enabling the user to define and edit the current cell's connection terminals.

In physical mode, the **Edit Terminals** button makes visible the current cell's contact terminals, if any have been assigned with the **subct** command. These terminals are automatically placed during association (if possible) however this command allows manual placement and editing of the properties of the terminals.

Terminals are moved and selected for editing using the same mouse and keyboard operations as in the **subct** command. One can click twice or drag the terminals to a new location. Multiple terminals can be selected and moved (unlike in electrical mode) by dragging over them with the mouse pointer, which displays a rectangle. The selected terminals are ghost-drawn and attached to the mouse pointer, during the move operation. In this state, pressing either the **Backspace** or **Esc** keys will deselect the terminals and abort the pending move. Terminals can not be deleted or created in this command, these operations must be done in the **subct** command.

If the **Shift** key is held while the user clicks on a terminal, or the user double-clicks on a terminal, including the case of a "move" to the same location, the **Terminal Edit** pop-up appears, just as in the **subct** command. The entries and effects are the same as are described for that command.

It is usually not necessary to place terminals manually. Exceptions are cells with ambiguous connection points. For example, suppose a cell contains a single resistor, with cell contacts "C1" and "C2" to the resistor. *Xic* will assign the physical locations of the terminals arbitrarily, which may not be the locations expected in a parent cell. For example, if the physical resistor is a vertical strip, a parent cell may expect C1 above C2, whereas *Xic* might have assigned the reverse. The user can move the terminals to the proper locations, bypassing the assignment in *Xic*, and the locations are made permanent when the cell is saved.

If association fails to place a terminal, or it is placed in the wrong location, then manual placement should be used. If the new location is over a conductor, that node/group association is assumed before the association operation (so it had better be correct, or association will not be correct).

The cell terminals have a `FIXED` flag which will be saved in cell files if set, the purpose of which is to prevent *Xic* from reassigning the physical location of the terminal. This flag will be set whenever the terminal is moved by the user. Once moved, the terminal should always remain in that location (which had better be correct for extraction/LVS to succeed).

The state of the flag is indicated by the check box in the **Terminal Edit** pop-up with label "Location locked by user placement". This flag can be set or unset with the check box.

When a terminal is placed, *Xic* searches through the conductor groups that touch the terminal for a suitable object to associate with the terminal. The object must touch the terminal, be on a Routing layer, and match the layer hint given to the terminal, if any. The hint layer can be supplied with the **Terminal Edit** pop-up, and is otherwise the last layer that the terminal was associated with (if any). If no object can be found that matches the hint, the hint is ignored and any Routing layer will be used. If an association is made, the layer name is printed near the terminal marker. If not, no layer name will be printed, and the terminal no longer has a hint. If the cell has not yet been associated, the layer name label may not appear. The actual association will be made the next time the cell is processed, which occurs when entering most of the extraction commands. In particular, activating the All Terminals or **Cell Terminals Only** check boxes is a benign way to force a recalculation of all associations. Better still is the **Do Extraction** button at the bottom of the panel.

Find Terminal
The **Find Terminal** button brings up the **Node (Net) Name Mapping** panel, which can also be brought up from the electrical side menu (see 7.11). In physical mode, the net naming operations are unavailable, however the facility for locating nets and terminals is always available.

Below the **Terminal** group are the **Select Unassociated** buttons. These can be used after extraction to identify the objects that failed to associate. These are objects in physical mode that have no identified electrical counterpart, and vice-versa.

The command works in physical and electrical modes. Display windows will highlight the appropriate unassociated objects for the window's display mode.

The highlighting is removed on a deselect operation, with the menu button or otherwise. Mostly, the objects are simply selected, however objects such as physical devices use other highlighting methods.

Groups/Nodes
Highlight nets and net objects (object groups) that are not associated between layout and schematic.

Devices
Highlight device symbols and layout elements that are not associated between layout and schematic.

Subckts
Highlight subcircuit symbols and placements that are not associated between layout and schematic.

This functionality is also available from the **!ushow** text-mode command.

16.9.2 The Net Config Page

This page provides control over aspects of net identification and naming.

At the top of the page is an input area and four check boxes. These relate to the net naming capability (see 16.5) using text labels in physical mode.

Net label purpose name
This group provides a text entry area and an **Apply** button. The **Apply** button must be active for the control to have any effect.

The text entered is the name of the purpose to be assumed for net naming labels. This tracks the PinPurpose variable. Pressing **Apply** will set the variable to the text name provided, which can be empty. Pressing **Apply** again will unset the variable reverting to the default purpose name of "`pin`", but the text in the entry area will be retained.

If the purpose name is set to an empty string, the "`drawing`" purpose is assumed. One could equivalently give the name explicitly. This is not really recommended as it can be inefficient. In this case, the label, and presumably its metal, would both reside on the same layer with the default purpose. For example, a label on a routing layer named "`M2`" placed over an object on `M2` would name the net containing the object. For efficiency, it is better to use a separate purpose. In the default case, the label would be on a layer purpose pair named "`M2:pin` in this example.

Net label layer

This works similarly to **Net label purpose name**, though sets the PinLayer variable. This is set to a layer name (or layer-purpose pair name) on which all net labels must reside. When set, the purpose mechanism is not used. This is for compatibility with older cell libraries, and is otherwise not recommended.

Ignore net name labels

If this check box is checked, existing net name labels will be ignored. This is probably only useful for debugging. The IgnoreNetLabels variable tracks the state (set or not set) of this check box.

Find old-style net (term name) labels

Earlier releases of *Xic* recognized "term labels" in the layout as identifying the conductor groups associated with cell terminals. These are labels, optionally created by the user. They must be created on a conducting layer, and placed over an object on the same layer.

If this box is checked, *Xic* will search for these labels, and treat them as net labels. These labels are formally identical to net labels if the **Net label purpose name** is applied as an empty string, or the purpose name "`drawing`". In this case, the search for term labels, which is a separate search from the search for net labels, would be redundant.

This check box links to the set/unset status of the FindOldTermLabels variable.

Update net name labels after association

When checked, net name labels will be updated, and new net name labels possibly created, after association completes. The label text is obtained from corresponding electrical net names.

WARNING: don't use this feature unless you understand the potential consequences. It is essential that association be correct when labels are created or updated. Incorrect labels in a layout will prevent correct association and will cause LVS errors. These would probably need to be removed or corrected by hand.

To use this, once LVS passes for a cell, one can clear extraction with the **Clear Extraction** button, check this check box, then press **Do Extraction**. The layout will now contain the new and updated labels. Uncheck this check box, and save the cell. This should only be done as a final step when creating a new cell, not while the cell is likely to change.

The presence of the net labels should reduce the time needed to associate the cell. Otherwise the added labels are redundant.

The UpdateNetLabels variable tracks the state (set or not set) of this check box.

Merge groups with matching net names

If two physically unconnected conductor groups have the same logical net name (see 16.5), if this box is checked the groups will be logically merged and treated as a single group. This allows successful top-level LVS of cells containing split nets. Below the top level, split nets are detected by other means so checking this box is not required for successful LVS if the top-level contains no split nets.

The group names that apply are obtained from net name labels, or from cell terminals that have been placed by the user. By default, net name matching is case-insensitive, though this can be changed with the NetNamesCaseSens variable. The name matching also treats as equivalent various subscripting delimiters, as listed in the description of the Subscripting variable.

The MergeMatchingNamed variable tracks the state of this check box.

Via Detection

Below the check boxes mentioned above is the **Via Detection** group. This controls how vias are searched for and identified. As most layouts contain large numbers of vias, the algorithms used to detect vias can have a significant impact on extration time. The default settings provide the least testing and the speediest performance, however they assume that a certain layout methodology has been followed. If all vias in the layout are separate cell instances, and the via masters contain patches of the two conductors along with the via layer, then the default settings will always apply. **If this is not the case, it is possible for connections to be missed, the user must understand the rest of this section**.

The recognition algorithm is as follows.

1. An object on a via layer is found, usually by iterating through the spatial database over a region.
2. If the object is not a box object, and the **Assume convex vias** check box is checked, it is replaced temporarily by a box of half the width/height of the bounding box. This is for efficiency – the geometrical computations are much faster for box objects. In practice, a via is almost always a square, but other shapes, mostly circular approximations, are occasionally seen. In particular, in superconductive electronics we often use circular vias concentric with circular Josephson junctions. If the check box is not set, the via shape is decomposed into a trapezoid list.
3. We look for objects on the upper and lower conductor layers whose intersection intersects the assumed via shape with nonzero area. If found, a connection is indicated.

The initial "grouping' phase establishes networks of metal objects in the cells, which contact by proximity or through vias. The next "extraction" phase modifies the network to account for device connections, and establishes connections to and between subcircuits. In this process, "wire-only" subcircuits are logically flattened into the parent cells. There are two things to keep in mind about this flattening process.

1. Only conductors are promoted into the parent cell. In particular, via material is not promoted, and must be accessed through the original cell hierarchy.
2. Consider a via cell, consisting of metal caps and a via pattern. The metal caps are connected, trivially. When the metal caps are promoted, they will tie together any contacting metal in the parent cell. Thus, we never have to recognize the via again, it has imposed its connectivity constraint and logically disappeared.

Connecting to and between subcells is a complicated and labor intensive operation. We have to iterate through all wire nets of all subcells, and test for connectivity. Connectivity can occur by direct contact, or, conceivably by a via. However, if we know that all of the vias are separate cells as described, we know that they have all been flattened away, and the hugely expensive test of looking for via connections between nets can be skipped entirely. If in the original layout a via (cell) is making the connection, in extraction the proximity test will discover the promoted, shorted via caps, and make the connection.

What if not all vias are in separate cells? In theory, the via material, and the two connected metal objects, may each occur in any subcell at any hierarchy level. In the most general case, we would need to search the entire hierarchy depth for via material, which can be painfully slow. However, other rules can apply. For example, to make a contact, one could place a square of via material on the current cell, over the metal layer intersection area of nets contained in the cell or subcells. If this method is used to connect between subcircuits, then this test must be enabled, however the search depth can remain at zero. If the via material is found in a subcell, then the search depth would have to be set appropriately.

The **Via Detection** group contains the following elements.

Assume convex vias

This applies when checking connectivity through a via during extraction. When set, vias that are not rectangular are assumed to be convex polygons, and connectivity testing is performed in a small rectangular region near the center of the bounding box. This is specifically for circular vias, as found in superconductive electronics. This simplifies testing and might speed extraction when circular vias are present. It should **not** be used if vias can take arbitrary polygonal shapes. This will have no effect on rectangular vias.

This sets/tracks the state of the ViaConvex variable.

Via search depth

If we have intersecting areas of top and bottom conductor, and we are searching for an area of via material that would connect the two metal objects, this sets the depth in the current cell hierarchy to search. The default is zero, indicating to search the current cell only. Generally, layout methodology can easily ensure that this value can be safely zero, but there may be cases that require extraction where such methodology was not practiced. In such a case, where the methodology is completely unknown, this value should be set to a large number (internally it is limited to 40, the maximum cell hierarchy depth) which will ensure that all via-induced connections are found. This can dramatically increase extraction time.

This tracks the value of the variable ViaSearchDepth, and defaults to zero if the variable is not set.

Check for via connections between subcells

By default, it is assumed that connections between subcells will be made by touching metal only. This includes the case where the metal is from a flattened wire-only cell, as would be provided by via cells as described previously. One can easily adapt layout methodology where this is true. Otherwise, this box can be checked, which will cause explicit testing for the presence of vias between subcircuit nets. This is a very expensive operation.

This tracks the whether or not the ViaCheckBtwnSubs variable is set.

Ground Plane Handling

The **Ground Plane Handling** group controls how the extraction system treats a ground plane. The is only required if the technology file defines a ground plane layer, which is unlikely to be true in semiconductor processing. The ground plane handling features were included specifically for Josephson junction process support, but can be applied to other technologies should the need arise.

Assume clear-field ground plane is global

If a clear-field ground plane has been identified in the technology file, when this box is checked, all areas of this layer are assigned group 0, the ground group. When not checked, only the largest area group in the top-level cell is assigned group 0. This tracks the GroundPlaneGlobal variable.

Invert dark-field ground plane for multi-nets
If a dark-field ground plane layer has been identified in the technology file, if this box is checked, the ground plane layer will be polarity inverted internally for extraction purposes. The inverted layer will be used to establish connectivity. This tracks the state of the GroundPlaneMulti variable.

Inversion method menu
When using an inverted ground plane, this menu provides a choice of methods. The default is to invert in each cell, then clip out the area occupied by subcells. The second choice will create the inverted layer in the top-level cell only, using the entire hierarchy as the source for geometry to invert. The third choice is similar, but will create the inverted layer in each cell, using as the source all geometry in that cell and its subcell hierarchy. This tracks the state of the GroundPlaneMethod variable.

16.9.3 The Device Config Page

This page contains controls that correspond to device-related extraction variables.

Device Block
Pressing the **Device Block** button produces a drop-down list of device blocks from the technology file, plus three additional buttons: **New**, **Delete**, and **Undelete**. The device blocks are listed in order of their definition, as the block name followed by the prefix, if any. Pressing **New** or any of the device block name entries brings up a text editor loaded with the indicated block, or empty for **New**. The text for the device block can be entered into the editor. Adding a block with the same name and prefix (or lack of a prefix) as an existing block will overwrite the existing block. Saving with the **Save** button in the editor will update an existing block or add a new block to the internal device list. The **Save** button in the editor *does not* save to disk. The **Save Tech** command can be used to generate a new technology file which will contain the new block, or the **Save As** button in the editor can be used to save the block as a file.

To delete a device block, press the **Delete** button in the **Device Block** menu, then select a device block from the same menu. That block will be removed from the menu. The name will disappear from the menu, and it is removed from consideration in extraction. The block can be restored with the **Undelete** menu entry, but only one deletion is remembered.

Don't merge series devices
If checked, series-connected devices will never be merged during extraction, overriding any `Merge` directive in the corresponding device blocks of the technology file. This tracks the state of the NoMergeSeries variable.

Similar devices may be series-merged if the common connection has no other connection. It is occasionally useful to suppress merging to individually measure the components of segmented devices, or in cells where the common contact may not have a connection that is currently in scope.

Don't merge parallel devices
When checked, parallel-connected devices are never merged during extraction, overriding any `Merge` directive in the device blocks of the technology file. This tracks the state of the NoMergeParallel variable.

If is occasionally useful to suppress parallel merging to individually measure segments of a multi-component device.

Include devices with terminals shorted
By default, if an extracted device is found to have all terminals shorted together, it is ignored. If this check box is checked, such devices are kept, allowing their parameters to be reported. This tracks the state of the KeepShortedDevs variable.

Don't merge devices with terminals shorted
When including devices with all terminals shorted, checking this box will prevent such devices from being merged as parallel devices, if parallel merging is enabled for the device type. This tracks the state of the NoMergeShorted variable.

Skip device parameter measurement
When this box is checked, device parameter measurement will not be performed during extraction. This may save time if the user is interested only in topology. This tracks the state of the NoMeasure variable.

Use measurement results cache property
When this box is checked, the extraction system will read and update (creating if necessary) the measures property (property number 7106) which is used to cache measurement results (see 16.7). The measurement of device parameters can be time consuming, and the caching can speed up the extraction process significantly. However, using the measurement cache may require user intervention to maintain coherency. If a device layout changes, the user will have to manually update the cache in order to obtain updated parameters. With this box not checked, the default condition will force actual computation of device parameters, and avoid all use of the caching mechanism. This is appropriate while a cell is under development, to avoid cache coherency issues.

The UseMeasurePrpty variable tracks the state of this check box.

Don't read measurement results from property
This setting is ignored unless **Use measurement results cache property** is checked. When this box is checked, the extraction system will not read the measures property (property number 7106) which is used to cache measurement results. When measurement results are required, they will be computed. The property will still be updated, after association, provided that **Use measurement results cache property** is set. Thus, by setting this check box and forcing association, one can get a fresh set of measurement results into the measures properties.

The NoReadMeasurePrpty variable tracks the state of this check box.

The remaining controls apply to the numerical solver used to extract resistance and (microstripline) inductance.

The default mode of the solver is to divide the device body into a grid such that the number of grid cells is fixed, independent of device size. This gives a predictable and constant measurement time per device, however it may provide less accuracy than other methods.

One can also choose to use a fixed grid size, in which case physically larger devices will take longer to extract, but computed values may be more accurate.

A third choice is to tile the structure, if possible, by using the largest grid such that all body and contact boundaries fall on grid. This is likely to provide the best accuracy if tiling succeeds.

Set/use fixed grid size
If the check box is checked, the solver will use a fixed grid size as given in the numeric entry area. When checked, other controls in this group are grayed and their states ignored. This tracks the state and value of the RLSolverDelta variable.

Try to tile
When checked, the solver will try to use a grid that falls on every edge of the contacts and device body. This tracks the state of the RLSolverTryTile variable.

Maximum tile count per device
When tiling is enabled, this entry area will set the maximum number of tiles allowed in a device. This tracks the state of the RLSolverMaxPoints variable, and defaults to 50,000.

Set fixed per-device grid cell count
This entry area supplies a number of grid cells to use per device. In this mode, the time required for extraction is close to constant, independent of device size. This mode is used when not tiling, and not using a fixed grid size. This tracks the state of the RLSolverGridPoints variable, and the default value is 1000;

16.9.4 The Misc Config Page

This page contains some miscellaneous settings and controls, to be described. The topmost controls on this page apply to cell handling.

Cell flattening name keys
This group contains a text entry area and an **Apply** button. When the **Apply** toggle button is in the pressed state, the FlattenPrefix variable is set to the text in the entry area. When the **Apply** button is not pressed, the text in the text area is retained, but the FlattenPrefix variable is unset.

This is one means by which the user can force a physical cell to always be logically flattened (see 16.4) in the extraction system. It is a bit archaic, as the extraction system will do most required flattening automatically, and use of the flatten property is the recommended way to force instance flattening.

Extract opaque cells, ignore OPAQUE flag
Checking this will cause the extraction system to ignore the OPAQUE flag applied to subcells, and attempt to extract the contents as for a normal cell. This tracks the state of the ExtractOpaque variable.

Be very verbose on prompt line during extraction
When set, during extraction lots of progress messages are displayed on the prompt line. However, this can actually slow down the process considerably, particularly if running remotely. By default, the messages are much more terse, and there is very little progress indication.

Global exclude layer expression
This group provides an **Apply** button and text entry area. The text entry area should contain the name of a layer, or layer expression. When the **Apply** button is pressed, the GlobalExclude variable will be set with the text from the entry. When the **Apply** button is unset, the text is ignored, and the GlobalExclude variable is unset.

If an expression is given and active, any object that touches an area where the layer expression is "dark" will be ignored in extraction, as if it didn't exist.

For example, one might create a layer named "`NOEX`". Then, if the layout contains features that should be ignored by the extraction system, one can enclose the features in `NOEX` boxes, and if "`NOEX`" is entered into the text entry area and the **Apply** button pressed, the marked features will be ignored.

The **Association** group contains settings that apply during the association operation, when electrical and physical matching occurs.

Don't run symmetry trials in association
When checked, the association computation will not break symmetry and run symmetry trials. This is mostly for debugging. The NoPermute variable tracks the state of this check box.

Logically merge physical contacts for split net handling
When set, some additional association logic is employed to detect and account for split nets in instance placements. A "split net" is a logical net consisting of two or more disjoint physical conductor groups. The disjoint parts of the net are connected when instances are placed, through parent cell metalization. If the schematic shows the net fully connected in the master, LVS will fail on the parent unless this check box is checked.

The MergePhysContacts variable tracks the state of this check box.

Apply post-association permutation fix
Checking this check box enables additional association logic. It applies when there is perfect topological matching between layout and schematic, but LVS is failing due to different permutations of permutable subcell contacts being assumed in the electrical and physical parts. Checking the box will enforce the electrical permutation on the physical solution, which will allow LVS to pass if the permutation difference was the only issue.

This should no longer be needed, as the two-pass association algorithm in current use should resolve these cases automatically. This check box should therefor not be checked in general, but it is possible that it might allow successful LVS in some obscure case.

The SubcPermutationFix variable tracks the state of this check box.

Maximum association loop count
This sets the maximum number of "loops" allowed around the association logic. There is no reason to touch this unless directed by Whiteley Research. The MaxAssocLoop variable tracks this entry.

Maximum association iterations
This sets the maximum number of iterations allowed per loop. There is no reason to touch this unless directed by Whiteley Research. The **MaxAssocIters** variable tracks this entry.

16.10 The Net Selections Button: Path Selection Control Panel

The **Net Selections** button in the **Extract Menu** brings up the **Path Selection Control** panel. This panel enables extraction-specific selection modes for groups, nodes, and connected conductor paths (wire nets). It is separate and distinct from the normal object and subcell selection. Command buttons in the panel replace menu buttons and modes found in other commands in earlier releases of *Xic*, in particular the group/node selection found in the **View Extraction** mode, and the **Show Paths** and **Quick Paths** commands found in the **Extract Menu** of *Xic* releases 3.1.4 and earlier are now available from this panel.

There are three basic selection modes available, which are set from the top row of buttons in the panel. Similar to normal selections, one clicks on an object in a drawing window. The object must be on a visible and selectable layer. Other selection specifications as found in the **Selection Control Panel** apply as well. In particular, one can choose the types of objects that are selectable, and the search-up mode. In search-up mode, layers are searched from bottom to top, rather than the default top to bottom, in physical mode. This affects the conductor chosen if the user clicks over more than one conductor layer.

Select Group/Node

When active, clicking on a conducting object in the current cell in a physical window will highlight all objects of the current cell in that conductor group. In electrical windows displaying the same cell, the corresponding node connections and wires will be highlighted. Clicking on a connection point or wire in an electrical window will highlight that node, and also highlight the corresponding group in physical windows.

Physical objects are "conducting" if the `Conductor` keyword was applied (directly or by inference) to the layer of the object.

If the **Select Path** button is also pressed while in this mode, the conducting path, as it recurses into subcells, will also be shown in physical windows.

Pressing the **p** key will toggle the state of the **Select Path** button and the display of recursive conductor paths.

With the mouse pointer in a physical window, typing a group number followed by **Enter** will switch to the display of that group and corresponding node. Similarly, in an electrical window, entering a node name or number followed by **Enter** will switch the display to the entered node and corresponding group. However, entering a name will probably trigger all kinds of accelerators, including those for this command, so there is a trick. Type a single or double quote character, followed by the node name. The quote character will inhibit recognition of accelerators. Since the keypress buffer length is only 16 characters, long node names can't be entered in this manner, the equivalent node number can be entered or some other method used to select the node.

In electrical mode, the command works with the **Node Mapping Editor** when visible. The currently selected node will always be highlighted in the node list panel of the editor. Selecting a node in the editor will highlight that node/group in the display windows.

Select Path

When the **Select Group/Node** button is also pressed, physical windows will display the conducting path of the currently selected group descending into subcells. Otherwise, this button initiates a different command for displaying conductor paths recursively. This mode is available in physical mode only. Clicking on a conducting object will highlight the conductor path containing that object. There is no selection or indication of the corresponding electrical node, nor will clicking in an electrical window have any effect in this mode. The clicked-on object need not be in the current cell (as is required for group/node selection), but must be within the search depth. The path generation algorithm makes use of the extraction system, and observes extracted devices and exclude directives as provided to the extraction system.

Only one path can be shown at a time. Clicking on another object will rebuild a path from the second object, erasing the original path, or it is possible to select a sub-path, it that feature is enabled.

If a dark-field ground plane is used, clicking on the painted areas (holes in the ground plane) will select the ground group, as will clicking on any other object which is connected to ground (group 0).

"Quick" Path

This command is similar to the **Select Path** command, but does not use the extraction system, except for establishing conducting layers and connections through vias. In particular, there is no information about devices and other extraction constraints established at higher levels. It may be useful for tracing wire nets, while skipping the sometimes lengthly extraction operation.

The **"Quick" Path** algorithm, unlike **Show Paths**, will ignore layers that are set invisible.

Since extraction is not used, there is no concept of devices, so that results may not be as expected, and not be as seen with the **Show Paths** mode. For example, consider MOS devices. Since, the source and drain are connected to a common area of the "active" layer, which is (usually) a `Conductor`, the simple algorithm used in this mode will interpret the source and drain as being connected together, since it does not recognize the MOS device. As a consequence, all wire nets are likely shorted together in this mode!

In order to get meaningful results, it may be necessary in this case to temporarily remove the `Conductor` keyword from the active layer. This can be accomplished with the **Tech Parameter Editor** in the **Attributes Menu**.

The remaining buttons and controls in the panel provide options or modes while the selections are active.

"Quick" Path ground plane handling
This menu applies only to the **"Quick" Path** selection mode, and sets the ground plane handling method. This tracks the setting of the QpathGroundPlane variable. If a dark-field ground plane (GroundPlaneClear keyword) has been specified in the technology file, the implied connectivity to ground is similar to that in force for the extraction system. There are three choices for handling the ground plane.

Use ground plane if available
This is the default. If an inverted ground plane has already been created and is current, it will be used when determining paths. If the ground plane does not have a current inversion, the absence of the layer will imply a ground contact, as in extraction without the MultiNet keyword. This choice avoids the sometimes lengthly inversion computation, but makes use of the inversion if it has already been done. The inversion is performed during extraction.

Create and use ground plane
If the extraction system would use an inverted ground plane, it will be created if not already present and current. The path selection will include the inverted layer.

Never use ground plane The **"Quick" Path** mode will never use the inverted ground plane.

Search path depth
This control and associated buttons apply when the **Select Path** or **"Quick" Path** modes are in effect. It determines the depth to recurse to when the conductor path is being constructed. If 0, only objects in the current (top-level) cell will be considered. The depth can be entered directly, or by clicking the up/down buttons, or by pressing the **0** or **All** buttons.

While the command is active, the expansion depth can also be changed with the -, +, **n**, and **a** keys. These decrement, increment, set to 0, and set to maximum, the depth, respectively.

When the depth changes, the path, if one is being shown, will be redrawn, if possible (the original object must be above the new depth).

"Quick" Path use Conductor
If this check box is not checked, only objects on layers with the Routing attribute applied will be considered for inclusion in the extracted path. If checked, objects on layers with the Conductor attribute will be allowed. The Routing attribute implies Conductor, but may be more restrictive.

The QpathUseConductor variable tracks the state of this check box.

Blink highlighting
Accelerator: h
When this box is checked, the highlighting in physical windows will blink. When unset, the highlighting will use the static highlighting color. Associated highlighting in electrical windows will always blink.

With a path being displayed, pressing the **h** key will toggle the blinking status.

Enable sub-path selection

This check box enables sub-path selection while in the **Select Path** or **"Quick" Path** modes.

When a path is displayed, the user can click on two objects in the path, and only the "sub-path" connecting the two objects will be highlighted. If the two objects are connected in multiple ways, the algorithm will select one (which may not be the most direct). If **Shift** is held while clicking on an object in the path, the object will be deselected and not considered as part of the path. This can be used to coerce a desired sub-path. When a sub-path is displayed, clicking on any non-selected object will display the full path containing that object.

Load Antenna file

Accelerator: f

This button applies to the **Select Path** mode only. Pressing this button will load a previously-generated antenna report file (from the **!antenna** command) for the current cell, and ask the user for a net number found in the file. The conductor path corresponding that that net number will be highlighted.

Pressing the **f** key while in **Select Paths** mode will also query an antenna report file in a similar manner.

To trapezoids

Accelerator: t

Pressing this button will decompose the geometric objects which comprise the currently shown physical conductor path into trapezoids. This has no effect on "real" objects in the database or in the extraction system, only the temporary objects used to display the selected path.

This can be useful in conjunction with the sub-path selection capability, to enable breaking a path by deselecting parts of an object that are separate as trapezoids. It may also be useful as a prelude to the **Save** operations in some cases.

Pressing the **t** key with a path displayed will also convert the path to trapezoid representation.

Save path to file

Accelerator: s

If a physical conductor path is being displayed, this button enables saving the objects that comprise the path to a native cell file. Only the selected objects will be exported. If the path has been converted to trapezoids, the trapezoid representation will be exported. Pressing the button brings up a small pop-up where the user can give a cell/file name. The resulting file can be read into *Xic* at a later time for further processing, or for conversion to another file format.

By default, the via layers are not included in the file, only the conductors. The two check boxes below the button allow saving the vias and other associated layers as well.

Pressing the **s** key with a path displayed will also save the path to a file in a similar manner.

Path file contains vias

This check box applies when the **Save path to file** button is used. When checked, the via objects that connect layers will be included in the generated path. If not checked, only the metal layers that constitute the path will be included in the file. The via layers are those that have the Via keyword defined in the technology file. The file will included the objects on the via layers, clipped to the intersection area of the two associated conductors.

Path file contains via check layers

This check box applies when **Save path to file** is used, and the **Path file contains vias** check box is checked.

The Via keyword line in the technology file contains an optional layer expression, which must be "true" for an actual connection to be indicated. For example

```
Via SBST MET1 DIFF&PPLS
```

This line would indicate that the layer containing this line forms a via between conductors SBST and MET1 only in the presence of layers DIFF and PPLS.

When this check box is checked, the file will contain the layers needed for the checking expression (DIFF and PPLS), clipped to the via layer objects. If not checked, the file will contain only the vias that meet the check criteria, but the layers needed for checking (DIFF and PPLS) will not appear.

With this box checked, the file can be loaded into *Xic* and extraction run, and the (single) net will be completely identified. This may not be the case if check layers are missing, and certainly won't be the case if via layers are omitted.

The two via inclusion check boxes track the state of the PathFileVias variable. If this variable is set as a boolean (i.e., to no value), then vias will be included, and check layers will not be included. If the variable is set to any text, the check layers will also be included.

16.10.1 Resistance Measurement

The **Resistance Measurement** buttons allow the user to measure the resistance between two points of the currently highlighted path.

Caveat: This is a new capability. The algorithm seems to have difficulty with some, usually complex, paths, meaning that a "pivot too small" or other error message will appear indicating lack of a solution.

All layers used in the path should have a sheet resistance specified. If no sheet resistance is specified, a value of 1 ohm/square is assumed. The sheet resistance can be specified directly with the Rsh keyword, or can be obtained if Rho or Sigma and Thickness have been given. If Rsh is not given, the value is taken as 1e6*Rho/Thickness, where Rho has units of ohm-meter and Thickness is given in microns. The conductivity Sigma is equal to 1.0/Rho. These keywords can be set in the technology file, or with the **Tech Parameter Editor** in the **Attributes Menu**.

To perform a measurement, the **Define Terminals** button should be used first to define two terminal locations. With the button pressed, drag mouse button 1 to define a rectangular area over some part of the displayed path. A box will be shown. Note that one must drag, with the mouse button pressed, to define the terminal area. Simply clicking has no effect. Repeat the process over another part of the displayed path, and a second box will be shown. These boxes represent the equipotential terminal areas assumed in the solver.

Once the terminals have been defined, pressing the **Measure** button should display the measured resistance on the prompt line. Diagnostic messages from the solver will be printed in the console window.

The algorithm does not include contact resistance between different metal layers.

16.11 The Device Selections Button: Show/Select Devices

The **Device Selections** button in the **Extract Menu** brings up the **Show/Select Devices** panel, from which devices can be made visible, and certain operations can be performed. There are three basic control groups.

The top control group contains a window which lists all of the devices extracted from the physical layout of the current cell. The listing has three columns. The **Name** and **Prefix** columns provide the values supplied in the technology file device block for the device. The third column gives the range of index values assigned for the device instances extracted. Each instance of a device has a unique index in this range.

The list is actually shown in response to pressing the **Update List** button. This will perform extraction/association on the current cell, if necessary, and list the devices found. With entries listed, the buttons above the listing become active. These buttons allow devices to be highlighted in the display windows.

Devices are highlighted is all windows showing the physical layout of the current cell. In addition, the corresponding electrical devices are also highlighted in windows showing the electrical schematic of the current cell.

Lines in the listing can be selected by clicking on the text. The buttons and other controls above the listing have the following functions.

Show All
All devices will be highlighted in the drawing windows.

Erase All
Erase all device highlighting in the drawing windows.

Show
The devices corresponding to the current selection in the list will be highlighted in the drawing windows. These are the devices that match the **Name** and **Prefix** selected, and whose indices are matched in the **Indices** entry text.

Erase The devices corresponding to the current selection in the list will be un-highlighted in the drawing windows. These are the devices that match the **Name** and **Prefix** selected, and whose indices are matched in the **Indices** entry text.

Indices
This is a text entry area where the user can provide a list of index integers and ranges to specify the index values of devices to highlight or un-highlight with the **Show** and **Erase** buttons. If the entry contains no text, all indices are used. The text consists of space or comma separated integers, or ranges of integers where the minimum and maximum values are separated by a hyphen (minus sign). For example: "`1,2-5,7,9-12`".

The second basic control group appears below the devices list, and enables devices to be selected by clicking on the device structure in the drawing windows. The command mode in initiated by pressing the **Enable Select** button. When in this mode, clicking on a device in a physical window showing the current cell will apply blinking highlighting to the device. The corresponding electrical device (if any) will also be shown with blinking highlighting in windows showing the electrical schematic of the current cell. In such windows, electrical device symbols can be clicked on, which will select the corresponding physical device.

Only one device can be selected at a time.

When the mode is active, the two check boxes to the right of the **Enable Select** button become active. When checked, information about new selections will be presented.

Show computed parameters of the selected device
When this box is checked, when a device is selected, the parameters extracted for the device will be printed on the prompt line. The format of the output is defined in the device block following the Cmput keyword.

Show elec/phys comparison of selected device

When this check box is active, clicking on a device will show a comparison of the extracted parameters and the corresponding electrical values for the device obtained from the schematic.

If one clicks on a device with the **Shift** key held, the electrical device properties will be set from the parameters extracted from the corresponding physical device. In an electrical window showing the device symbol, the device property labels will appear or change when the properties are set or updated.

The physical device must be specified in a device block, and have at least one parameter with the LVS keyword specified.

The third basic control group allows electrical parameters for the current layer to be measured for a rectangular region. If also allows rectangular regions to be painted, and can be used as an alternative to the **box** command in the side menu. Unlike the other two basic control groups, this group is only active in physical mode.

Electrical information is applied to layers in the with the Rsh keyword for resistance (or alternatively Rho or Sigma along with Thickness), the Capacitance keyword for capacitance, and the Tline keyword for transmission line parameters. The electrical specifications may be added or edited with the **Edit Tech Params** command in the **Attributes Menu**.

When the **Enable Measure Box** button is pressed, the command mode becomes active. The user can drag or click twice in physical windows to define a rectangular area. This area will be outlined with a highlighting box. During the creation, and after the box is created, the electrical parameters, if any, from the current layer are applied to the box dimensions, and the electrical parameters are displayed.

Once the box is created, pressing the **Paint Box** button, or pressing the **p** key, will paint the highlighting box with the current layer, creating a box object in the cell.

This is useful for creating simple rectangular resistors, for example, as the readout facilitates creating the proper size for the desired resistance. The command can also be used to measure the values of existing rectangular resistors.

The mouse operations can be repeated, as long as the command remains active. Only one highlighting rectangle is available at a time.

16.12 The Source SPICE Button: Update From SPICE File

The **Source SPICE** button in the **Extract Menu** allows electrical information in a schematic to be updated or generated by reading a SPICE file. Pressing **Source SPICE** brings up a small pop-up containing an entry area for the name of a SPICE file to read, and three check boxes. The entry area is active as a drop receiver, so that the **File Selection** panel (or another file manager program) can be used to locate the file, and the name can be dragged into the entry area. The **Go** button will actually initiate the operation.

Node name mapping is turned on after the operation completes. Since a schematic produced in this way has every node name defined by a terminal, using the defined names, which correspond to the original SPICE file, is convenient.

The three choice buttons are:

all devs

If set, all devices in the cell which match a name in the SPICE file will be updated. If not set, only the devices that have names that were set explicitly by the user (by applying a name property) are updated.

create
If set, devices specified in the SPICE file that are not found in the schematic are created. If not set, only the properties of existing devices are updated. If the current cell is empty, **create** is taken as set.

clear
If set, the electrical part of a cell is cleared before reading the SPICE input. This implies **create**.

If **create** is set, or the target cell is empty, this command will create a schematic hierarchy from the SPICE file. The function may be used as follows: open a new cell and go to electrical mode. Use the **Source SPICE** button to read in a SPICE file. The devices and subcircuits referenced in the file will be arrayed in the drawing, each with the appropriate properties applied. Named terminals are placed at each device contact point, which establish connectivity (wires are not used). The drawing can be used for simulation or any purpose just as a schematic entered in the standard way. The created schematic can be modified by the user to replace the named terminals with wires and reset the device locations, to make a "real" schematic that is aesthetically decent.

Subcircuits are created as needed. They must be written out later (e.g., with the **Save** command). If a file exists in the search path with the same name as a subcircuit, it is ignored, as the subcircuit cells are created internally. When writing, therefore, it is possible to replace an existing cell file, but the previous version is retained with a "`.bak`" extension.

Devices are instantiated as needed, and given an assigned name from the SPICE file.

If **create** is not active, no new devices or subcells will be instantiated in a non-empty cell, though devices in the drawing with names which match those in the SPICE file will have their properties updated. Properties of existing devices are updated whether or not **create** is active. Similarly, if a subcircuit already exists, its devices will be updated, but no new devices will be created in the subcircuit.

If **all devs** is not set, only devices that have been assigned a name by the user will have properties updated. Devices with internally assigned names are skipped. This is to avoid problems due to the fact that internally assigned names will change when the circuit is edited, and updating from an out-of-sync SPICE file could be a disaster.

If **clear** is set, then the electrical part of the cell and subcells will be cleared before the SPICE information is read. This ensures that the cells contain only information supplied in the SPICE file.

In order to determine if a semiconductor device is a p-type or n-type, *Xic* will look for a corresponding model in the source file, or the model library if not found. If still not found, if the model name starts with "n" or "p", or if the model name contains "n" but not "p" or *vice-versa*, *Xic* will infer the type. If none of this succeeds, the operation is aborted, and the user must provide access to the device model.

Xic will also test the consistency of MOS models defined in the technology file (used when extracting physical data) and the MOS model assumed for use in the device library (usually the `device.lib` file). If the node count differs, a warning will be issued. The warning indicates that LVS will fail. See the discussion in the description of the DeviceKey global device library property in B.8.1 for more information.

All "dotcards" that are not otherwise handled are written verbatim in the top-level schematic as labels on the SPTX layer. Recall that the labels on this layer are "spicetext" labels (see 7.9.4), so that the label text is included in SPICE output generated from the cell. Labels will not be created if a label with matching text already exists.

There is no inclusion of text from `.include` or `.lib` lines or similar, these become labels on the

SPTX layer in the top-level schematic.

Subcircuit calls that have no subcircuit definition will be written as spicetext labels, and a warning will be given. It is likely that they are resolved at simulation-time through `.include` or `.lib` inclusions.

All `.model` lines found in the SPICE source are written to a file in the current directory named "*cellname*`_models.inc`", where *cellname* is the top-level cell name. In the schematic of the top-level cell, a label is created on the SPTX layer containing a `.include` line for the models file (if the label does not already exist). Models that are defined within subcircuits are given a new hierarchical name to ensure uniqueness.

Nested subcircuit definitions are handled by assigning a new hierarchical name to the subcircuit (cell) which is unique.

Parameter definitions from `.param` lines, subcircuit definitions, and subcircuit instances are applied verbatim as param properties of cells and instances, or as labels on the SPTX layer for `.param` lines, within the cell corresponding to the subcircuit where found.

Although parameterization of subcell instances is allowed and works fine for simulation and other purposes, these parameters are effectively ignored in LVS. LVS requires that a unique master be created for each instantiation parameter set, and the parameterized instances be replaced by instances of the appropriate master.

Each of the option buttons has a corresponding **!set** variable. If the variable is changed while the pop-up is visible, the pop-up will be updated. Conversely, changing the state of the option buttons will set or unset the corresponding variables. The pop-up check box will be checked if the corresponding variable is set. The names of the corresponding variables are given in the table below.

all devs	SourceAllDevs
create	SourceCreate
clear	SourceClear

There are two additional variables that are used by this command. These specify the names of the ground and terminal devices, as provided by the device library file, that this command will use. Generally, it is not necessary to set these variables, as the defaults should always be appropriate. The user may, however, prefer to use an alternative terminal style, or may have a custom device library with different names for these devices from those found in the `device.lib` file distributed with *Xic*.

SourceGndDevName

This variable specifies the name of the ground terminal device to use. If not set, the name "`gnd`" will be assumed. If this variable is set to a name, a ground device of that name must appear in the device library file.

SourceTermDevName

This variable specifies the name of the terminal device to use. If not set, the name "`tbar`" will be assumed, if that name is found for a terminal device in the device library. If not found, the name "`vcc`" will be assumed. If this variable is set to a name, that name must match the name of a terminal device in the device library file.

16.13 The Source Physical Button: Update Electrical From Physical

The **Source Physical** button in the **Extract Menu** will update the electrical part of a design from parameters extracted from the physical part. The command works by writing a temporary SPICE file from the physical database, then updating the electrical database from the SPICE file. When the **Source Physical** button is pressed, a small pop-up appears, which is similar to the pop-up seen with the **Source SPICE** command, but has no text entry area, and has an additional **Depth** choice menu which sets the depth into the hierarchy to process. The **Go** button initiates the operation.

Node name mapping is turned on after the operation completes. Since a schematic produced in this way has every node name defined by a terminal, using the defined names, which correspond to the physical group numbers, is convenient.

The first three check boxes have similar functions as in the **Source SPICE** command. The remaining check box enables inclusion of wire-net capacitors.

all devs
If set, all devices in the cell will be considered for updating If not set, only the devices that have names that were set explicitly by the user (by applying a name property) are updated.

create
If set, missing devices are created. If not set, only the properties of existing devices are updated.

clear
If set, the electrical part of a cell is cleared before updating. This implies **create**.

include wire cap
If set, capacitors that represent routing net capacitance will be updated, or created if they don't exist and **create** is set. These capacitors are given a special name prefix "`C@NET`" which has significance to *Xic*, i.e., it identifies them as routing capacitances. The capacitors are added between the wire nets and ground. In order for wire capacitance to be computed, the Capacitance keyword must be supplied in the technology file for the routing layers.

ignore labels
From some tools, cell terminals may be indicated by the presence of a label on a Routing layer, positioned such that the label reference point touches an object on the same layer. Such labels, if found, will be used to generate a terminal list for the top-level cell in the extracted hierarchy, if the existing electrical cell contains no terminals (or the electrical cell doesn't exist). If this box is checked, such labels will always be ignored.

Each of the option buttons has a corresponding **!set** variable. If the variable is changed while the pop-up is visible, the pop-up will be updated. Conversely, changing the state of the option buttons will set or unset the corresponding variables. The pop-up check box will be checked if the corresponding variable is set, unless the variable name has a "No" prefix, in which case the logic is reversed. The names of the corresponding variables are given in the table below.

all devs	NoExsetAllDevs
create	NoExsetCreate
clear	ExsetClear
include wire cap	ExsetIncludeWireCap
ignore labels	ExsetNoLabels

16.14 The Dump Phys Netlist Button: Dump Physical Netlist

The **Dump Phys Netlist** button in the **Extract Menu** creates a netlist file from the physical connectivity information in the current cell. Upon pressing this button, a small pop-up appears, which provides a number of format options. The options include the names from the `PnetFormat` blocks in the format library file, if any. The format library provides a mechanism for user-specified formatting of netlist output. The supplied `xic_format_lib` file contains a formatter for the Cadence DEF (Design Exchange Format) format, as well as a simple example format.

There are three built-in format choices: **net**, **devs**, and **spice**. Any combination of the formats can be selected, and the output will contain a block for each selected format, for each cell.

In addition, there are a number of options which modify the presentation. These include **list all cells** and **list bottom-up**, which apply to all formats, and **show geometry** and **include wire cap**. The latter options are enabled when **net** and **spice** are enabled, respectively, or when a format library choice is active.

The format options will be described in more detail below. Below the format check boxes there is a **Depth** choice menu which allows setting of the depth into the hierarchy to process. The user is given the option of creating the netlist to an arbitrary depth in the hierarchy. If the given depth is greater than zero, the subcells above the indicated depth will also be added to the file. If "all" is selected, the full hierarchy will be output.

Below the depth menu is a text entry area for the name of the file to be generated. The default name is the base name of the current cell, suffixed with "`.physnet`", to be created in the current directory. The entry area is sensitive as a receiver for drag/drop.

Any combination of the four format options may be selected. The states of the option check boxes track the status of the variables described below. The listing from the **Dump Phys Netlist** command will have a field of output for each selected format, from each cell. Pressing the **Go** button will produce the output file.

The format option check boxes are described below. The first two are options that apply to all formats.

list all cells
: Subcells that are wire-only or otherwise internally flattened or ignored are normally not listed. If set, these cells are included in the listing, which may be useful for debugging.

list bottom-up
: When the depth is larger than zero, this check box controls the ordering of cells in the file. When selected, the deepest cells (the "leaf cells") are listed ahead of their parent cells, thus the current cell will be listed last. When not selected, the listing is top-down. The current cell is listed first, followed by subcells.

The next three rows of option check boxes specify the internal formats and options for these formats.

net
: A netlist consisting of the terminal names associated with each conductor group is generated.

show geometry
: If this is selected, the **net** part of the output file will include a listing of the physical objects that comprise the wire net. This includes objects from the present cell, and objects that have been

promoted from wire-only subcells. The objects may not exactly correspond to the physical objects, for example if the Conductor Exclude directive is given. The objects are listed in a modified CIF syntax, where units correspond to internal database units.

devs
A list of extracted devices, with information about the device, including Measure results, is generated.

spice
A list of the SPICE lines for extracted devices which have a Spice specification in the device block is generated.

include wire cap
When active, the SPICE listing will contain capacitors for nonzero computed wire net capacitance. These capacitors are given a special prefix "`C@NET`" which has significance to *Xic*, when applying LVS. The capacitors are added between the wire nets and ground. In order for wire capacitance to be computed, the Capacitance keyword must be supplied in the technology file for the routing layers.

ignore labels
From some tools, cell terminals may be indicated by the presence of a label on a Routing layer, positioned such that the label reference point touches an object on the same layer. Such labels, if found, will be used to generate a terminal list for the top-level cell in the listed hierarchy, if the existing electrical cell contains no terminals (or the electrical cell doesn't exist). If this box is checked, such labels will always be ignored.

devs verbose
This check box is active when the **devs** check box is checked. When checked, it enables printing of additional information in the device report in the output file. At present, it will print information about the individual components of multi-component (series or parallel merged) devices.

Additional option buttons, if any, correspond to formats specified in the format library file. If selected, a text block containing the output from the format generator will be appended to the file, for each cell. The following are available from the stock distribution format library file.

DEF
This uses a formatting script in the `xic_format_lib` file to generate DEF output. DEF is a common portable netlisting format. See the comments in the `xic_format_lib` file in the startup directory for more information.

phys-example
This uses a formatting script in the `xic_format_lib` file to generate output in simple example format.

Each of the option buttons that correspond to an internal format or option (not the formats from the library) has a corresponding **!set** variable. If the variable is changed while the pop-up is visible, the pop-up will be updated. Conversely, changing the state of the option buttons will set or unset the corresponding variables. The pop-up check box will be checked if the corresponding variable is set. The names of the corresponding variables are given in the table below.

list all cells	PnetListAll
list bottom-up	PnetBottomUp
net	PnetNet
show geometry	PnetShowGeometry
devs	PnetDevs
spice	PnetSpice
include wire cap	PnetIncludeWireCap
ignore labels	PnetNoLabels
devs verbose	PnetVerbose

16.15 The Dump Elec Netlist Button: Dump Electrical Netlist

The **Dump Elec Netlist** button in the **Extract Menu** creates a netlist file from the electrical connectivity information in the current cell. Upon pressing this button, a small pop-up appears, which provides a number of format options. The options include the names from the `EnetFormat` blocks in the format library file, if any. The format library provides a mechanism for user-specified formatting of netlist output. The supplied `xic_format_lib` file provides a formatter for Cadence DEF (Design Exchange Format), and a simple example format.

There are two built-in format choices: **net** and **spice**. Any combination of the formats can be selected, and the output will contain a block for each selected format, for each cell. In addition, there is one format option, **list bottom-up**, which applies to all formats.

The format options will be described in more detail below. Below the format check boxes there is a **Depth** choice menu which allows setting of the depth into the hierarchy to process. The user is given the option of creating the netlist to an arbitrary depth in the hierarchy. If the given depth is greater than zero, the subcells above the indicated depth will also be added to the file. If "all" is selected, the full hierarchy will be output.

Below the depth menu is a text entry area for the name of the file to be generated. The default name is the base name of the current cell, suffixed with "`.elecnet`, to be created in the current directory. The entry area is sensitive as a receiver for drag/drop.

Any combination of the format options may be selected. The states of the option check boxes track the status of the variables described below. The listing from the **Dump Elec Netlist** command will have a field of output for each selected format, from each cell. Pressing the **Go** button will produce the output file.

The format option check boxes are described below. The first option applies to all formats.

list bottom-up
When the depth is larger than zero, this check box controls the ordering of cells in the file. When selected, the deepest cells (the "leaf cells") are listed ahead of their parent cells, thus the current cell will be listed last. When not selected, the listing is top-down. The current cell is listed first, followed by subcells.

The next two option check boxes specify the internal formats.

net
A netlist consisting of the terminal names associated with each wire net is generated.

spice

A SPICE listing is generated.

Additional option buttons, if any, correspond to formats specified in the format library file. If selected, a text block containing the output from the format generator will be appended to the file, for each cell. The following are available from the stock distribution format library file.

DEF

This uses a formatting script in the `xic_format_lib` file to generate DEF output. DEF is a common portable netlisting format. See the comments in the `xic_format_lib` file in the startup directory for more information.

elec-example

This uses a formatting script in the `xic_format_lib` file to generate output in simple example format.

Each of the option buttons that correspond to an internal format or option (not the formats from the library) has a corresponding **!set** variable. If the variable is changed while the pop-up is visible, the pop-up will be updated. Conversely, changing the state of the option buttons will set or unset the corresponding variables. The pop-up check box will be checked if the corresponding variable is set. The names of the corresponding variables are given in the table below.

list bottom-up	EnetBottomUp
net	EnetNet
spice	EnetSpice

If the variable CheckSolitary is set with the **!set** command then warnings are issued if nodes are encountered with one connection only.

16.16 The Dump LVS Button: Test Layout vs. Schematic

The **Dump LVS** (Dump Layout Vs. Schematic) button in the **Extract Menu** compares the netlists obtained from the physical and electrical data for the hierarchy of the current cell, and lists topological and electrical differences. When the **Dump LVS** button is pressed, a small pop-up appears, which contains a field for setting the name of the output file, and has provision for setting the depth into the hierarchy to compare. The default name for the output file is the base name of the current cell, with a "`.lvs`" extension, and this will be written in the current directory unless a path is given to the file name. Entering 0 for the depth compares the current cell only, 1 compares the current cell and immediate subcells, and so on. The user is given a chance to view the output file upon completion.

If computed wire capacitance is included in the electrical data, the capacitors will be recognized by virtue of having a special name prefix "`C@NET`"" and treated specially. Unlike other devices, there is no corresponding physical device. If found, the values will be compared with the corresponding computed net capacitance in the physical data, and an error will be reported if the two numbers differ by 1 percent or more. Wire net capacitance is considered only for the capacitors that are found in the electrical data, i.e., if they are missing no error is generated.

When the LVS data are printed out, the hierarchy of the electrical (schematic) part is used as the basis. This means that

1. any physical structures that are not connected to the top-level cell (directly or indirectly) and are not represented in the schematic are ignored.

2. the reverse is not true: anything in the schematic that doesn't have a physical counterpart is an error.

Thus, the schematic is favored, as anything not in the schematic and not connected physically is considered to be a "test structure" and is generally ignored. One of the reasons for this behavior is the potential existence of test cells and structures that might contain real devices or circuits, which aren't connected to anything but are used for process analysis. Generally, one would expect these to be ignored for LVS purposes.

However, unconnected physical subcells (cell instances) that contain extracted devices or subcircuits are explicitly checked for and listed. If the **fail if unconnected physical subcells** check box in the **LVS** panel is checked, the presence of unconnected physical subcircuits will force LVS failure of the cell. This check box tracks the state of the LvsFailNoConnect variable.

16.16.1 Parameterization Limitation

Although electrical subcircuit instance parameterization is allowed and works fine when generating simulation files for SPICE, it is ignored in LVS. The LVS system implicitly assumes that a cell and its instances are precisely similar, that an instance of a cell is in all respects defined by the master cell of the instance. Instance parameterization is therefor not recognized (but parameters defined in the cell itself are fine).

One has similar issues with parameterized physical cells. With parameterized cells, a unique master is created for each unique set of instantiation parameters used in the design. The template cell "instance" is not really an instance of the template cell, but is actually an instance of a master created for a particular parameter set.

Within LVS, each physical template master would correspond to an electrical master, and likewise there would be correspondence between instances. Presently, all of this must be configured manually. Work is ongoing to fully support parameterization through SPICE, physical and electrical cells, and LVS, in a transparent manner.

16.16.2 Using the nophys Property

The nophys property can be applied to electrical devices and subcircuits, causing them to be ignored in the extraction system, notably in LVS. Devices that have no physical representation, such as voltage sources, have this property set by default.

By "ignoring" these devices, the device terminals are considered as open circuits. However, there are times when it would be useful to consider these devices as shorted. For example, suppose that one wishes to include parasitic series inductance in a resistor during simulation. However, this inductance would cause LVS to fail, since the series inductor added to the schematic has no explicit physical counterpart.

It is possible to configure the nophys property to indicate that when the electrical netlist is generated for use by the extraction system, the flagged nophys devices will be forced to have all terminals connected to the same net, i.e., the terminals are effectively shorted together. Thus, the inductor in the example above, if given this property, would disappear properly during LVS. However, when generating a SPICE netlist for simulation, these devices will be included in the netlist.

There are a number of aspects to using the nophys property.

1. The cached internal electrical netlist can be in one of two states, respecting shorted nophys or not.

If there are no shorted nophys devices, both representations are the same. Functions that require one representation or another will invisibly rebuild this when needed.

2. All operations in the extraction system, including the **Extract Menu** functions and extraction script functions, will respect the shorted nophys property. This includes the SPICE format listings from electrical data in the **Extract Menu**.

   The **run**, **deck**, and other similar functions in the side menu that relate to SPICE simulation will *never* respect the nophys property, these devices will be treated as other devices.

3. In electrical mode, nophys devices are shown in a different color on-screen (yellow by default, the "Terminal Color").

4. The **Property Editor** will query the user whether to set the shorted option when a nophys property is added.

5. There is a **Use nophys** button in the **Node Name Mapping** editor from the side menu. This button selects whether or not to respect shorted nophys devices in the node listings. Shorted devices can obviously change the node numbering.

6. The string stored in the nophys property can either be "`nophys`" or "`shorted`". *Xic* sets these values according to the state.

7. There is an `IncludeNoPhys` script function which can be used with the existing electrical netlist access functions to provide the nophys recognition state desired.

16.16.3 LVS Output File Format

For each cell comparison, the LVS system reports four levels of success.

CLEAN
: Everything was measurable and matched.

PASSED - AMBIGUITY
: There were device parameters which could not be compared, but all comparisons that were done matched.

  In the electrical schematic, if component values are parameterized (i.e., use a token defined in a `.param` line or similar), or perhaps use *WRspice* shell expansion, the value was unavailable. In earlier releases, a value was available only if it was a numeric constant. *Xic* now provides limited parameter substitution during LVS (see below).

PASSED - PARAM DIFFS
: There were device parameters that differ outside of the tolerance between electrical and physical. So actually, only the circuit topological check passed.

FAILED
: Differences in circuit topology were detected.

The overall result for the run is the lowest level in this hierarchy reported for any cell.

The parameter database and substitution code was imported from *WRspice* for use during LVS and elsewhere. However, not all capability can be provided.

1. Parameters given in subcircuit call lines are ignored in LVS, making LVS meaningless if these are given in the schematic. Parameterized instances must be remastered to unique master cells for the current LVS system.

2. There is presently no support for macros defined in `.param` lines. However, single-quoted expressions are fully supported, all math operations and all relevant functions are available.

Parameter expansion works as follows:

1. When an LVS run starts, the parameters defined in the top-level cell as param properties, and all parameters defined in `.param` lines found in labels on the SPTX layer in the top level cell, are placed in a table.

   In addition, the labels on the SPTX layer are searched for `.option` lines, and these lines are searched for a parhier option, and if found, its setting is saved. This option can be set to one of "`global`" (the default if not found) or "`local`".

2. When comparing devices in the top-level cell, the parameter table is used to parameter substitute the value and param property strings. The resulting string should provide numerical values for comparison to the extracted physical values.

3. When comparing in a subcell/subcircuit, the subcircuit param properties and `.param` labels are tabulated as for the top-level cell. This is merged with the top-level table, and is used to expand the param and value property strings of devices in the cell.

   If the `parhier` option was found, and it was set to `local`, then parameters defined in the subcircuit table will override conflicting definitions in the top level table. If `parhier` wasn't found or was set to `global`, the reverse is true – top-level definitions will override conflicting definitions in the subcell.

The output file produced by LVS contains a block of lines for each cell in the hierarchy where there is both electrical and physical information. Each block may contain several tables, which provide information about the cell and the electrical/physical associations. These tables are described below.

Conductor group and electrical node mapping

Xic assigns an integer to every physical wire net (called a "group") and to every electrical wire net (called a "node", as in SPICE). These numbers are in general different. In addition, a node may have a text name that was assigned by the user.

This table displays the group to node and node to group mappings. The entries under the "node" heading display the internal node number in parentheses, followed by the actual node name (which will simply be the number again if no node name was assigned).

Formal terminal group associations

In this listing, the first column is the terminal name, the second column is the associated group number (you can find the electrical node from the group/node mapping table). If association failed for the terminal, i.e., *Xic* was unable to place the terminal in the layout, the word "UNINITIALIZED" will appear in the third column. This will cause LVS to fail for the cell.

Physical device associations

If the physical cell contains devices, then this table will appear. Each device of a given type in the schematic is assigned a number, and devices extracted from the physical layout are assigned a (generally different) number.

An entry appears for each device extracted from the physical data. The first line for the device contains the device name and the physical index number. If the device has an electrical counterpart, the electrical device type (same as the physical name) and electrical name are printed on the same line, following a colon. The electrical name uses the SPICE convention. This line is followed by a listing of the device terminals, one line per terminal. The terminal name and group number are to the left of the colon. If the group is associated, the associated electrical node number (in parentheses) and name are given to the right of the colon. These lines are optionally followed by a listing of extracted parameter values for the device. The actual format and displayed parameter set is defined in the corresponding device block in the technology file.

Physical subcircuit associations

If the physical cell contains subcells, then this table will appear. The first column gives that name of a subcell found in the physical cell. If the cell is actually an array, each element of the array will be listed, with the array indices in parentheses following the name. The second column is the internal index assigned to the subcell for physical mode. If there is a corresponding electrical subcell, the electrical subcell type and name will be shown, following a colon. The subcell type is the same as the physical subcell name. The subcell name is the subcircuit name in the schematic. This usually follows the SPICE convention of using 'X' as the leading character. This is followed by a listing of the subcircuit terminals, one line per terminal. The physical group numbers in the cell and subcell are printed to the left of the colon. To the right of the colon, the electrical node numbers (in parentheses) and names in the electrical cell and subcell are printed. If a group number is not associated, the corresponding node number is shown as "-1" and the node name is "???".

Checking for unconnected physical subcircuits

Physical subcells that contain extracted devices or subcells that have no connection to the circuit may be present. Since the electrical hierarchy is used for recursion, these are not detected in the traversal, since they have no representation in the schematic and no connection to the circuit. However they are checked for explicitly. If any such subcells are found, they will be listed, but otherwise ignored, unless the LvsFailNoConnect variable is set, in which case LVS will fail on the presence of such cells.

Checking per-group/node terminal references

For each group/node association, *Xic* will compare the list of terminals connected to the physical group with the list of terminals connected to the electrical node. The lists should be the same. This header may be followed by a list of terminal referencing errors. Possible errors are device, subcircuit, and formal terminals that are connected to the physical group but not the electrical node, or vice-versa. Such errors will cause LVS to fail for the cell.

Summary

The final table, which always appears, is the summary. This will report nonassociations, and will indicate whether the cell passed or failed the LVS test.

A pass indication is reported for a cell if all of the following are true:

1. All electrical nets, devices, and subcircuits are associated, meaning that *Xic* has identified the corresponding object in the physical layout.

2. No associated physical device or subcircuit is connected to an unassociated group.

3. No unassociated physical device or subcircuit has a connection to an associated group other than the ground group (0).

4. Parameter value comparisons between corresponding electrical and physical devices match.

Note that having unassociated physical groups, devices, or subcircuits does not automatically cause failure. Unassociated groups (random pieces of conductor material) do no harm, but all groups connected to associated devices or subcircuits must be associated (have a corresponding node in the schematic). It is also possible to have unassociated physical devices or subcircuits, but none of these can have a connection to associated groups other than the ground group (the ground group is used when a ground plane layer is specified). Thus, the physical layout can have structure not represented in the schematic, but only if this structure is topologically disjoint from the associated circuit.

16.17 The Extract C Button: Capacitance Extraction

The **Extract C** button in the **Extract Menu** brings up **Cap Extraction** panel which controls the interface to an external program used for capacitance extraction.

16.17.1 The Capacitance Extraction Interface

The interface uses an external program to extract capacitance values between conducting features in the layout. The interface supports the following capacitance extraction programs:

1. The *FasterCap* program from `FastFieldSolvers.com`. This commercial program is recommended for users with capacitance extraction as an important workflow element. The auto-refinement capability provides the best accuracy with the least amount of setup.

2. The *FastCap* program from Whiteley Research. The interface also provides a crude, linear panel refinement capability which can be used with this free version of *FastCap*, which is available from the Whiteley Research free software archive. We will refer to the Whiteley Research program as *FastCap-WR* to distinguish it from the MIT original.

The interface generates a unified list file, which is compatible with the programs listed above. It is **not** directly compatible with the original MIT *FastCap* program, or its derivatives, that require multiple input files. There are accessory programs `lstpack` and `lstunpack` available which convert between the formats, so the MIT *FastCap* can be used in a two-step flow involving unpacking.

This is the second generation capacitance extraction interface. The original capacitance extraction interface, found in releases 4.0.8 and earlier, was quite a bit more complicated. The present interface affords at least the following simplifications:

- The interface presently takes material from the current cell as input, there is no need to select and save things into the interface.
- There is no "dataset name", the file names use the current cell name as a base name.
- The output file is always a unified list file, there is no "old format" support except via the separate `lstpack` and `lstunpack` utilities.
- There is no graphical "partition editor" as *FasterCap* does not need external refinement.

The new interface, however, is much more flexible and powerful than the original interface.

- There is no longer a fixed assumption that layers are planar, or that layer ordering must begin with a conductor and alternate with insulators. Layers can appear in any order, and any layer can be planarizing, or not. If a layer is planarizing, it will have variable thickness such that the top surface is in one plane.
- There is a new layer-sequencing engine (see 12.8) that is also used by the **Cross Section** display command (in the **View Menu**), as well as for inductance/resistance extraction. Thus, the cross section display will always faithfully represent the assumptions used in the interface. Layer ordering is basically that shown in the layer table, though Via layers are allowed to be out of sequence (likely for drawing visibility reasons). The "real" position of a Via layer can be obtained from the layers it references.
- Geometry is taken from the current cell, to all levels of the hierarchy. There is provision for use of a special masking layer. If this layer is found in the layout, geometry will be clipped to the patterning on this layer.
- The substrate is now more accurately included in the calculation, taking into account the actual thickness and lateral extent.

Geometry Construction

By default, a layer named "`FCAP`" will serve as the masking layer. If, however, the FcLayerName variable is set to a layer name, that layer will provide the masking function. We will refer to this layer as the "mask layer".

If no objects are found on the mask layer, all geometry in the current cell will be treated in the interface. Be advised that the capacitance extraction will very rapidly become untenable if too much geometry is included. The interface itself is not designed to handle large object collections, though it will remain snappy while generating files that may take weeks to run. More than 100 objects is probably pushing things. The effective area of interest (AOI) is the bounding box of the current cell.

If objects are found on the mask layer, then the mask layer pattern is **and**ed with the other layers, and the resulting geometry is processed by the interface. The bounding box of the mask layer patterning becomes the effective AOI, which can be much smaller than the cell bounding box. In any case, the AOI bounds all geometry in the problem.

The geometry is as shown in the drawing window, though geometry is saved in an internal representation that removes any overlap of objects in the original layout. Outside of the AOI, and above all geometry and below the substrate, vacuum (relative permittivity of 1.0) is assumed.

The substrate is included in the calculation as follows. A variable named SubstrateThickness can be set to specify the assumed substrate thickness in microns. If not set, a thickness of 75 microns is assumed. Typically, the substrate thickness would be set in the technology file with the `SubstrateThickness` keyword, which sets the variable. It can be set interactively from the **Params** page of the **Cap Extraction** panel (see 16.17.2).

If the substrate has nonzero thickness:

The boundary of the substrate is taken as the AOI, bloated by the value given by the FcPlaneBloat variable. This is generally desirable to move the substrate edge effectively away from structures of interest. If not set, a value of 0.0 micron is assumed.

Interface panels will be created on the sides and bottom of the substrate when the input list file is generated. If a positive FcPlaneBloat is given, dielectric interface panels will also cover the top of the substrate outside of the AOI.

If the substrate has zero thickness:

This is obtained by setting the SubstrateThickness variable to 0. We attempt to treat the substrate as filling the infinite half space, though is is not clear how to convey this to *FasterCap*. Outside of the AOI, the substrate/vacuum interface extends to infinity. We approximate this with finite panels extending a distance given by FcPlaneBloat out of the AOI.

NOTE: the original interface made no attempt to deal with the substrate. This is reasonable, as the different substrate treatments should have little effect on results in most cases.

Note also that the FcPlaneBloat parameter extends the substrate only, and not the geometry. This is different from the original interface, which would also extend the dark-field layers. To effectively bloat the geometry as well as the substrate, one can use the `FCAP` layer in most cases to enlarge the AOI.

Technology File Setup

Setup parallels setup of the three-dimensional layer sequence database (see 12.8), which in turn follows setup of the extraction system (see 16.8). These sections should be consulted for detailed setup information, here we provide some supplemental information.

To the interface, there are two different materials:

conductors
: Conductors will have one of the following:

1. Any of the `Conductor`, `Routing`, `GroundPlane`, `GroundPlaneClear`, or `Contact` technology file keywords (or their aliases) applied. All of these implicitly give the Conductor keyword.
2. Any of the Rsh, Rho, Sigma, or Lambda keywords applied with a positive value.

insulators
: Insulators will have one of the Dielectric or Via keywords applied, and also the EpsRel keyword applied with a value of 1.0 or larger.

In addition, layers that are to be used in the interface as conductors or insulators must have all of the following:

- A Thickness keyword applied with a value greater than zero.
- Must be visible in the layer table.
- Must not have the Symbolic keyword applied.

The Dielectric technology file keyword was added to support this interface. This is intended to model an explicit capacitor dielectric, and differs from Via layers in the following ways.

- Unlike Via, it is not assumed to be dark field (but DarkField can be applied to the layer explicitly).
- Only one Dielectric keyword can appear per layer (multiple Via keywords are allowed).
- Stacking order is as shown in the layer table (Via layers are allowed to appear out of order).
- Dielectric layers are not planarizing by default, Via layers are.

The present interface can take layers in any order. This is in contrast with the original interface, that required layers to alternate conductor/insulator starting with a conductor, and ordering was obtained entirely from Via references and not the layer table order.

After all possible layers from the layer table are sequenced, layers that are not used in the extracted geometry are discarded. Note that dark-field layers are inverted, as we are interested in representing the physical material. Thus, for example no structure in a Via layer (i.e., no vias) in the layout implies the presence of a continuous film of insulating material, so the layer is actually present.

The same layer sequencer is used in the **Cross Section** command in the **View Menu**. The cross section display and the interface will always agree on the ordering and planarization of the layers. It is therefor a useful visualization tool when setting up the layers in the technology file.

The **Cross Section** command is also useful for finding errors. One possible error occurs when not planarizing, and a thin metal layer runs over the edge of a thicker dielectric layer. This will disconnect the metal between the two sides of the step, which will cause a failure in the extraction. Presently, the interface assumes that the number of conductor groups remains the same before and after 3D processing. The disconnection can be easily seen in the **Cross Section** view.

In addition to the layers that describe material geometry, the interface can make use of a masking layer. This allows only certain specified parts of the current cell to be evaluated. When present, geometry is clipped to objects on this layer before being processed in the interface.

By default, a layer named `FCAP` with purpose `drawing` is assumed for the masking layer. Such a layer should be defined in the technology file. It should be given a GDSII mapping to allow saving of work containing the layer to GDSII or OASIS files. As an alternative, the FcLayerName variable can be set to the name of another layer, which will instead provide the masking function.

Output File

The output file is a unified *FasterCap* list file. At the top of the file is a comment containing the layer sequence. For example:

```
* Layers          Plane       Thickness    EpsRel
* Substrate                   75.000       11.900
* Insulator CO    0.000       0.190        4.200
* Insulator VIA1  0.190       0.095        2.900
* Insulator VIA2  0.285       0.095        2.900
* Insulator VIA3  0.380       0.095        2.900
* Conductor M4    0.475       0.220
* Insulator VIA4  0.695       0.095        2.900
* Conductor M5    0.790       0.220
* Insulator VIA5  1.010       0.095        2.900
* Conductor M6    1.105       0.220
* Insulator VIA6  1.325       0.095        2.900
* Insulator VIA7  1.420       0.610        4.200
* Insulator VIA8  2.030       0.610        4.200
```

The `Plane` is the base elevation of the mask objects of a planarizing layer, that is, the top surface of the layer minus the layer thickness value. This field will be empty for non-planarizing layers.

16.17.2 The Cap Extraction Panel

This panel, brought up by the **Extract C** button in the **Extract Menu**, controls the interface to external capacitance extraction programs described above. The interface can also be controlled to a large extent with the **!fc** prompt line command.

The panel functionality is divided into three pages, selectable through the tabs along the top of the window. Common to all pages is a **Help** button, status line, and **Dismiss** button. The status line indicates the number of background extraction jobs currently running.

The Run Page

The **Run** page contains controls for running the supported programs, or creating unified list format input files for these programs. This is the default page, shown when the panel appears.

Run in foreground

At the top of the page is the **Run in foreground** check box. When checked, the program will run synchronously in the foreground, rather than asynchronously in the background. Aside from possibly being helpful when debugging problems, it is not clear that this mode has any value.

This check box sets, and is set by, the FcForeg variable.

Out to console

When the **Out to console** check box is checked, the program output will be printed in the console window, i.e., the shell window from which *Xic* is running. This is most useful with *FasterCap*, which iterates to a solution, and the user can verify that all is well by watching this output.

This check box sets, and is set by, the FcMonitor variable.

Show Numbers

When the interface is run to produce an input file, the mutually connected conducting shapes are identified, and each disjoint group is assigned a conductor number. These numbers are used in the input list file to specify the conductors, and in the output file to identify the capacitance matrix indices.

When this check box is checked, the conductor numbers will be shown on-screen, so that the user can easily determine the conductor numbers associated with the layout objects. Each number is shown as a cross-mark, which will appear at a corner of an object in the conductor group. The conductor number, and the layer name of the associated object, are printed next to the cross-mark.

Note that due to the possibility of clipping by an `FCAP` layer, objects that are connected in the cell may be disconnected when used in the interface. In that case, two or more marks may appear over the same object, or different objects that are touching. If this is confusing, one can use the **Layer Expression** panel from the **Edit Menu** to create temporary layers for visualization, consisting of the conducting layers **and**ed with the `FCAP` layer. The resulting shapes will have unique conductor number marks.

Run File
This button and adjacent text entry allows an arbitrary input file to be run by the capacitance extraction program currently configured. The text area should contain a path to a valid input file for the configured program. The program will run, and results will appear, as for a normal extraction run.

Run Extraction
This button will dump a temporary input file, run the program, and display the results. The result file is named *cellname-pid*`.fc_log`, where *cellname* is the name of the current cell, and *pid* is the process id of the spawned process used to run the program. The file contains listings of the input file produced by the interface and the output file produced by the program.

By default, the program is run in the background. The label at the bottom of the panel will indicate that the job is running. When complete, a **File Browser** window containing the result file will appear. While jobs are running in the background, one can continue using *Xic*.

If the FcForeg variable is set, from the **Run in foreground** check box or with the **!set** command, then the program will instead run in the foreground. In this case, the result file is named *cellname*`.fc_log`, and *Xic* will be unresponsive until the run completes.

Dump Unified List File
This button allows an input file to be generated, which is in a unified list format compatible with the supported programs. The default name for this file is *cellname*`.lst`, where *cellname* is the current cell name.

FcArgs
This text entry area can be given a string, which will be included in the argument list when the program is run with the **Run Extraction** button. This allows specialized command line options to be provided during the run, which the user may require. This entry field is tied to the FcArgs variable.

If the interface detects that *FasterCap* from `FastFieldSolvers.com` is being used, and this entry is empty, the default argument string

```
-b -a0.01
```

will be imposed. A "`-b`" option will always be added if missing from the *FasterCap* arguments list, as this argument is necessary for correct *FasterFap* operation in this mode. The "`-a`" option is almost always used, as it specifies auto-refinement, however it is technically not necessary and won't be imposed if not given, except in the case where no arguments are given at all.

Path to FasterCap or FastCap-WR
Near the bottom of the page is an entry area where the path to the executable program can be edited. This entry area displays and sets the FcPath variable.

The Params page

The upper half of this page provides entry areas for parameters used by the interface related to the substrate, plus a menu for choosing the units to use in the list file.

SubstrateThickness
This sets the assumed substrate thickness in microns. When the thickness is nonzero, the substrate bottom and sides are assumed to abut vacuum permittivity. When the thickness is set to zero, the substrate is assumed to completely fill the half-space below the extraction area.

This entry sets, and is set by, the SubstrateThickness variable.

FcPlaneBloat
This entry contains a length, in microns. If nonzero, horizontal dielectric/vacuum interface panels will extend outside of the area of interest (AOI, see 16.17.1) along the top surface of the substrate. The extension distance is the FcPlaneBloat distance.

When the SubstrateThickness is nonzero, the substrate bounding box, which is the AOI, will be bloated by this value before writing of the substrate bottom and side interface panels to the list file. This will move the abrupt dielectric change at the substrate edge away from the area of interest.

If the SubstrateThickness is zero the FcPlaneBloat distance should be large enough to represent "infinity", but making it too large will slow down computation. The model is approximating the entire half-space filled with substrate dielectric material.

This entry sets, and is set by, the FcPlaneBloat variable.

SubstrateEps
This entry supplies the relative dielectric constant asssumed for the substrate. This sets, and is set by, the SubstrateEps variable.

FcUnits
This is an option menu which is used to set the length units used in files produced by the interface. Choices are meters, centimeters, millimeters, microns (the default), inches, and mils. The selection, if not the default, will set the FxUnits variable. Similarly, setting the variable with the **!set** command will update the state of the menu. The choice currently in effect will be applied when input files are generated. The choice of units will not affect the computed capacitance.

The lower half of the page allows one to crudely refine the raw panels while being written to the list file. This is specifically for *FasterCap-WR*, which requires refined panelization for accuracy. The *FasterCap* program does not require external refinement, which is a major advantage. In fact, the refinement provided here should **not** be used with *FasterCap*, as it may interfere with *FasterCap*'s refinement.

The refinement is "crude" due to each refined panel being approximately the same size. If the size is small enough, sufficient spatial resolution for accurate capacitance calculation is achieved. This resolution is needed along edges, and at corners, where there are strong field gradients, but is gross overkill for most areas. Since the solving time is related to the total number of refined panels, this type of refinement is very inefficient with respect to memory use and execution speed.

The refinement works as follows. First, the interface computes the total area of all conductor and dielectric raw panels that would be output to the list file. This area is divided by the **FcPanelTarget** number provided by the user. This is a number approximating the total refined panel count that *FastCap-WR* will need to process. The solution time should be approximately the same for the same panel count, independent of the actual geometry. The square root of the divided area is used when writing the panels to the list file. The raw panels are subdivided so that no panel edge is longer than this value.

A number like 10000 is probably about right for the **FcPanelTarget** in providing decent accuracy in a reasonable execution time. Larger numbers provide more accuracy, but require larger files and have longer solution time. The list file will contain a line for each refined panel. The entry area will take numbers up to 1e6, which is probably unreasonable for a normal computer.

Enable
This check box will enable or disable the refinement. This should not be active when using *FasterCap*. When pressed, the **FcPanelTarget** entry will become un-grayed, and internally the FcPanelTarget variable will be set to the number shown in the **FcPanelTarget** entry area. When the **Enable** button is set inactive, the FcPanelTarget variable is unset. The **Enable** button state will reflect whether or not the FcPanelTarget variable is set.

FcPanelTarget
This entry area is seensitive only when the **Enable** check box is checked. It tracks the value of the FcPanelTarget variable, which can be set to a real value of 1e3 – 1e6. This will be the approximate number of refined panels generated in the list file.

The Jobs page

The **Jobs** page contains a list of running background jobs. Each entry provides the process identification number (PID), the name of the executing program, and the local date and time when started. Entries can be selected by clicking with the mouse.

When an entry is selected, the **Abort job** button below the list becomes un-grayed. Clicking this button will terminate the selected process. The user should consider that there is no confirmation and no ability to resume the run.

16.18 The Extract LR Button: Inductance/Resistance Extraction

The **Extract LR** button in the **Extract Menu** brings up the **LR Extraction** panel which controls the interface to an external program used for inductance and resistance extraction. Presently, the *FastHenry* program is supported. The interface should be compatible with any version of *FastHenry* or derivatives. A guaranteed-compatible version is distributed as free software on the Whiteley Research web site (`http://wrcad.com`).

16.18.1 The Inductance/Resistance Extraction Interface

The interface uses an external program to extract inductance and resistance values along conducting features in the layout. The interface currently supports the original *FastHenry* program from MIT, and (presumably) all input format compatible successor and derivative programs that can be run from a command line.

- **FastHenry-WR** from Whiteley Research. The program is available available in the free software archive on the Whiteley Research web site. This package has been updated to build easily with newer compilers. It has been extended to support superconductors, and incorporates features to accelerate computation.

- **FastHenry** from MIT. This is the original *FastHenry* three-dimensional inductance extraction program.

This is the second generation interface to *FastHenry*. The original extraction interface, found in releases 4.0.8 and earlier, was quite a bit more complicated. The present interface affords at least the following simplifications:

- The interface presently takes material from the current cell as input, there is no need to select and save things into the interface.
- There is no "dataset name", the file names use the current cell name as a base name.
- There is no longer a graphical "partition editor". This was too cumbersome. Instead, a simple automatic refinement provision is included. It is hoped that one day auto-refinement will be built into a *FastHenry* successor program, as was done in the *FasterCap* program from `fastfieldsolvers.com`.
- There is no longer a graphical terminal definition editor. Instead, a special layer is used to define terminal areas.

The new interface, however, is much more flexible and powerful than the original interface.

- There is no longer a fixed assumption that layers are planar, or that layer ordering must begin with a conductor and alternate with insulators. Layers can appear in any order, and any layer can be planarizing, or not. If a layer is planarizing, it will have variable thickness such that the top surface is in one plane.
- There is a new layer-sequencing engine (see 12.8) that is also used by the **Cross Section** display command (in the **View Menu**), as well as for capacitance extraction. Thus, the cross section display will always faithfully represent the assumptions used in the interface. Layer ordering is basically that shown in the layer table, though Via layers are allowed to be out of sequence (likely for drawing visibility reasons). The "real" position of a Via layer can be obtained from the layers it references.
- Geometry is taken from the current cell, to all levels of the hierarchy. There is provision for use of a special masking layer. If this layer is found in the layout, geometry will be clipped to the patterning on this layer.

Geometry Construction

By default, a layer named "`FHRY`" will serve as the masking layer. If, however, the FhLayerName variable is set to a layer name, that layer will provide the masking function. We will refer to this layer as the "mask layer".

If no objects are found on the mask layer, all geometry in the current cell will be treated in the interface. Be advised that the inductance/resistance extraction will very rapidly become untenable if too much geometry is included. The interface itself is not designed to handle large object collections, though it will remain snappy while generating files that may take weeks to run. More than 100 objects is probably pushing things. The effective area of interest (AOI) is the bounding box of the current cell.

If objects are found on the mask layer, then the mask layer pattern is **and**ed with the other layers, and the resulting geometry is processed by the interface. The bounding box of the mask layer patterning becomes the effective AOI, which can be much smaller than the cell bounding box. In any case, the AOI bounds all geometry in the problem.

The geometry is as shown in the drawing window, though geometry is saved in an internal representation that removes any overlap of objects in the original layout. Outside of the AOI, and above all geometry and below the substrate, empty space is assumed.

The geometric specification of conductors to *FastHenry* consists of "node" definitions, and the definition of "segments" that connect the nodes. The nodes are points in three-dimensional space, and touch or are enclosed in conducting objects. Once the node locations are assigned, the connecting segments are created. Nodes and segments that dead-end are removed, the remaining nodes and segments can be thought of as a three-dimensional SPICE net, where the segments represent inductors. We compute the overall inductance between different "terminals" of the network.

The interface works by tiling. This assumes that current can flow in any direction. A tile is a rectangular prism, with a node in the center, and a node at the center of each of the six faces. Segments, whose dimensions are set by the tile size, connect the central node with each of the face nodes. In order for adjacent tiles to make contact, the face nodes of the adjacent faces must coincide. This will be true if the conducting objects are decomposed into tiles properly, avoiding corners that are adjacent to an edge. The decomposition is performed along the X, Y, and Z directions. Tiles with a side longer than a given maximum dimension will be subdivided.

This is the appropriate way to handle three-dimensional current flow through constrictions such as vias, and account for penetration or skin depth. However, *FastHenry* was originally set up to handle long, thin conductors where the current flow is assumed to be in a fixed direction. Most of the examples use this approach, and use flags that internally subdivide the segments transverse to current flow. This provides suitable accuracy and good efficiency for certain types of problems.

When the full three-dimensional decomposition is used, the problem size can quickly grow to the point where *FastHenry* can not provide a solution in a reasonable amount of time. Approximations must be employed at this point to reduce segment count. To a much greater extent that for capacitance extraction, the user may have to intervene to fine-tune the process.

The ability to tile requires that the geometry be Manhattan. Support for non-Manhattan geometry is provided by first internally Manhattanizing non-Manhattan objects before processing. The granularity of the Manhattanization is controlled by the FhManhGridCnt variable or the corresponding text entry field in the **LR Extraction** panel **Params** page. The length of a Manhattan segment used to approximate non-Manhattan geometry must be larger than `sqrt(`*area_of_interest*`FhManhGridCnt)`. Using FhManhFridCnt values larger than the default 1000 will increase accuracy, but this can dramatically increase the segment count, and therefor *FastHenry* run time. When possible, non-Manhattan features should be avoided when using this interface to *FastHenry*.

Terminal Definition

Terminals are the assumed external contact points used when extracting the inductance matrix. Unlike capacitance extraction, inductance and resistance extraction requires terminal definitions, and the results will depend fundamentally on the terminal locations.

Terminals are specified by creating boxes or polygons and labels on special layers. The feature will define as equivalent all nodes that touch or are enclosed in the shape, for any Z coordinate (the features are in the X-Y plane).

Each terminal feature must have at least one overlapping text label on the same layer, that provides the terminal name. Terminals must resolve to pairs, where each pair is a "port", taking the inductance matrix as an N-port network. The pair is ordered, with one terminal being the "plus" terminal, the other the "minus" terminal.

This is all accomplished by adherence to the following rules.

1. The features (boxes or polygons) and labels which equivalence nodes and define terminals are created on special layers. The layer name is the same as the layer name of the conductor which provides the nodes. The purpose name is the special keyword "`fhterm`". Thus for example, for a metal layer named "`M1`" (which has the default "`drawing`" purpose) the corresponding special layer has the full layer-purpose pair (LPP) name "`M1:fhterm`". Such a LPP should be defined for each conducting layer in the technology file.

2. Terminal features must touch or enclose at least one node. Nodes can be found at the center of each edge of each tile. When connecting to the end of a metal strip, for example, the entire transverse width of the strip end should be enclosed in or touch the terminal feature, so that current flow is uniform.

3. Each terminal feature must have at least one overlapping text label on the same layer giving a terminal name.

4. Each terminal is one of a pair, the pair representing a port. The terminals of each pair must contact the same conductor group, i.e., be connected.

5. It is possible for a terminal to be used in more than one port, in which case the terminal will have more than one overlapping label.

6. Port-terminal association is by name. Name labels must follow these rules:

   (a) Terminal names consist of a port name and a suffix. If the name string contains punctuation or white space, the first occurrence of such is stripped, and the port name is taken as the characters to the left, and the suffix is taken as the characters to the right, of where the punctuation or white space resided. If there is no punctuation or white space, the port name is the name string with the rightmost character stripped, and the suffix is this character. The port name and suffix must each contain at least one printable character or a fatal error results. The port name is arbitrary, but must be unique among the ports.

   (b) Both terminals of a port must have the same port name, case sensitive. It is a fatal error if a terminal can not be paired.

   (c) Both terminals of a port must have different suffixes. The suffix is used only to order the terminals in the port. This is done using lexicographic ordering of the suffix strings. Beyond ordering, the suffix is ignored. It is a fatal error if the suffixes are the same.

   (d) Both terminals of a port must contact the same conductor group.

7. For each terminal feature, a list of intersecting nodes is created internally. The first node in the list is taken as the reference. If there are additional nodes, they are equivalenced to the reference node, using the *FastHenry* "`.equiv`" construct.

8. For each port, a *FastHenry* "`.extern`" construct is used to provide the reference nodes of the two terminals in order, followed by the port name.

Technology File Setup

Setup parallels setup of the three-dimensional layer sequence database (see 12.8), which in turn follows setup of the extraction system (see 16.8). These sections should be consulted for detailed setup information, here we provide some supplemental information.

To the interface, there are two different materials:

conductors
: Conductors will have one of the following:

1. Any of the `Conductor`, `Routing`, `GroundPlane`, `GroundPlaneClear`, or `Contact` technology file keywords (or their aliases) applied. All of these implicitly give the Conductor keyword.
2. Any of the Rsh, Rho, Sigma, or Lambda keywords applied with a positive value.

insulators
: Insulators will have one of the Dielectric or Via keywords applied, and also the EpsRel keyword applied with a value of 1.0 or larger.

In addition, layers that are to be used in the interface as conductors or insulators must have all of the following:

- A Thickness keyword applied with a value greater than zero.
- Must be visible in the layer table.
- Must not have the Symbolic keyword applied.

Conductor layers can have the following optional keywords defined. These control the filamentation of conductor layers carrying current in the plane of the substrate, in the direction normal to the substrate. This accounts for penetration or skin depth in planar areas of material, in the Z direction. Typically, for superconductors at least, lateral dimensions are much larger than film thicknesses, so the volume element refinement tends to keep the film thinkness unbroken in the Z direction. For accurate account of the penetration depth, this should be further subdivided, and filamentation is one way to accomplish this.

- A FH_nhinc keyword with an integer value greater than one can be applied. This is the number of filaments into which the segment will be divided, along the normal to the substrate. See the *FastHenry* documentation for more information about the `nhinc` parameter which can be applied to segment definitions.
- A FH_rh keyword with a real value different from the default value of 2.0 can be given. When the filament count is larger than 2, filaments have varying height so as to maximize the density at the film surfaces. This parameter sets the ratio of heights between adjacent filaments. A value of 1.0 means that all filaments have the same height. See the *FastHenry* documentation for more discussion and information about the `rh` parameter.

The conductor layers can be given a resistivity or conductivity with the Rho and Sigma keywords, respectively. Additionally, the Lambda parameter, which specifies the London penetration depth for superconductors, can be specified. This is for the convenience of *Xic* users in the superconducting electronics R&D community. In this case, Rho/Sigma specify the unpaired conductivity from the two-fluid model.

The Dielectric technology file keyword was added to support capacitance extraction. This is intended to model an explicit capacitor dielectric, and differs from Via layers in the following ways.

- Unlike Via, it is not assumed to be dark field (but DarkField can be applied to the layer explicitly).
- Only one Dielectric keyword can appear per layer (multiple Via keywords are allowed).
- Stacking order is as shown in the layer table (Via layers are allowed to appear out of order).
- Dielectric layers are not planarizing by default, Via layers are.

The present interface can take layers in any order. This is in contrast with the original interface, that required layers to alternate conductor/insulator starting with a conductor, and ordering was obtained entirely from Via references and not the layer table order.

After all possible layers from the layer table are sequenced, layers that are not used in the extracted geometry are discarded. Note that dark-field layers are inverted, as we are interested in representing the physical material. Thus, for example no structure in a Via layer (i.e., no vias) in the layout implies the presence of a continuous film of insulating material, so the layer is actually present.

The same layer sequencer is used in the **Cross Section** command in the **View Menu**. The cross section display and the interface will always agree on the ordering and planarization of the layers. This was not true with the original interface.

In addition to the layers that describe material geometry, the interface can make use of a masking layer. This allows only certain specified parts of the current cell to be evaluated. When present, geometry is clipped to objects on this layer before being processed in the interface.

By default, a layer named `FHRY` with purpose `drawing` is assumed for the masking layer. Such a layer should be defined in the technology file. It should be given a GDSII mapping to allow saving of work containing the layer to GDSII or OASIS files. As an alternative, the FhLayerName variable can be set to the name of another layer, which will instead provide the masking function.

Finally, the layers used in terminal definitions should be configured into the technology file. Generally, there is an *Xic* layer corresponding to each conducting layer, with a purpose name "`fhterm`" and the same base layer name. In order to save the layout with terminals as GDSII or OASIS, a GDSII layer mapping should be applied for these layers.

Output File

The output file is a *FastHenry* input file, with format as documented in the original *FastHenry* manual, potentially with the extensions for superconductivity support (`lambda` specifications) if the Lambda parameter is given to any layer used (making it a superconductor). Such constructs are **not** compatible with the original MIT *FastHenry*, but require the Whiteley Research version, or a derivative.

At the top of the file is a comment containing the layer sequence. For example:

```
* Layers           Plane        Thickness    EpsRel
* Substrate                     75.000       11.900
* Insulator CO     0.000        0.190        4.200
* Insulator VIA1   0.190        0.095        2.900
* Insulator VIA2   0.285        0.095        2.900
* Insulator VIA3   0.380        0.095        2.900
```

```
* Conductor M4     0.475        0.220
* Insulator VIA4   0.695        0.095        2.900
* Conductor M5     0.790        0.220
* Insulator VIA5   1.010        0.095        2.900
* Conductor M6     1.105        0.220
* Insulator VIA6   1.325        0.095        2.900
* Insulator VIA7   1.420        0.610        4.200
* Insulator VIA8   2.030        0.610        4.200
```

The `Plane` is the base elevation of the mask objects of a planarizing layer, that is, the top surface of the layer minus the layer thickness value. This field will be empty for non-planarizing layers.

Tips and Hints

- It is easy to generate input files that take an excessively long time to run. This is probably the most common pitfall. The run time increases with increasing segment count. The segment count is approximately proportional to the number of lines in the input file.
- Non-Manhattan features can be very bad news. These are "Manhattanized" by converting non-Manhattan edges into stepwise approximations. The large number of edges that can be produced by this may lead te excessive run time. The **FhManhGridCnt** entry in the **Params** page of the control panel can be decreased to reduce the number of segments.
- Too-few segments is also to be avoided, as the calculation may not be suitably accurate. The **Volume Element Refinement** check box in the **Params** page should always be checked. The entered number can be varied, the user should experiment. Larger values should provide better accuracy at the expense of longer computation time.
- One should not try to extract "too much", due to the FastHenry limitations. How much is too much depends on a lot a factors, the user should experiment to gain a feel for their process and computer hardware.

16.18.2 The LR Extraction Panel

This panel, brought up by the **Extract LR** button in the **Extract Menu**, controls the interface to external inductance/resistance extraction programs described therein. The interface can also be controlled to a large extent with the **!fh** prompt line command.

The panel functionality is divided into three pages, selectable through the tabs along the top of the window. Common to all pages is a **Help** button, status line, and **Dismiss** button. The status line indicates the number of background extraction jobs currently running.

The Run Page

The **Run** page contains controls for running the supported programs, or creating input files for these programs. This is the default page, shown when the panel appears.

Run in foreground

At the top of the page is the **Run in foreground** check box. When checked, the program will run synchronously in the foreground, rather than asynchronously in the background. Aside from possibly being helpful when debugging problems, it is not clear that this mode has any value.

This check box sets, and is set by, the FhForeg variable.

Out to console
When the **Out to console** check box is checked, the program output will be printed in the console window, i.e., the shell window from which *Xic* is running. It may be useful to verify that all is well by watching this output.

This check box sets, and is set by, the FhMonitor variable.

Run File
This button and adjacent text entry allows a compatible input file to be run by the inductance/resistance extraction program currently configured. The text area should contain a path to a valid input file for the configured program. The program will run, and results will appear, as for a normal extraction run.

Run Extraction
This button will dump a temporary input file, run the program, and display the results. The result file is named *cellname-pid*`.fh_log`, where *cellname* is the name of the current cell, and *pid* is the process id of the spawned process used to run the program. The file contains listings of the input file produced by the interface and the output file produced by the program.

By default, the program is run in the background. The label at the bottom of the panel will indicate that the job is running. When complete, a **File Browser** window containing the result file will appear. While jobs are running in the background, one can continue using *Xic*.

If the FhForeg variable is set, from the **Run in foreground** check box or with the **!set** command, then the program will instead run in the foreground. In this case, the result file is named *cellname*`.fh_log`, and *Xic* will be unresponsive until the run completes.

Dump FastHenry File
This button allows an input file to be generated, which is in a format compatible with the *FastHenry* program. The default name for this file is *cellname*`.inp`, where *cellname* is the current cell name.

FhArgs
This text entry area can be given a string, which will be included in the argument list when the program is run with the **Run Extraction** button. This allows specialized command line options to be provided during the run, which the user may require. This entry field is tied to the FhArgs variable.

FhDefaults This text entry allows the user to provide a string which will appear in a `.DEFAULT` line in the created *FastHenry* input file. See the FastHenry documentation for syntax and options. This entry field is tied to the FhDefaults variable.

FhFreq
This consists of three entry areas, which take the starting and ending evaluation frequencies for *FastHenry* runs, and the number of intermediate frequencies to evaluate. This corresponds to the `.Freq` specification line in *FastHenry* input files. The frequencies are given in hertz. If the third field in empty, then evaluation is at the specified frequencies only. This variable is tied to the FhFreq variable, which can also be set with the **!set** command.

Path to FastHenry
Near the bottom of the page is an entry area where the path to the *FastHenry* executable program can be edited. This entry area displays and sets the FhPath variable.

The Params page

This page provides entry areas for parameters used by the interface.

FhUnits
This is an option menu which is used to set the length units used in files produced by the interface. Choices are meters, centimeters, millimeters, microns (the default), inches, and mils. The selection, if not the default, will set the FhUnits variable. Similarly, setting the variable with the **!set** command will update the state of the menu. The choice currently in effect will be applied when input files are generated. The choice of units will not affect the computed inductance/resistance.

FhManhGridCnt
Value: real number 1e2–1e5.
When a non-Manhattan polygon is "Manhattanized" for *FastHenry*, it is converted to an approximating Manhattan polygon. The value in this entry is used to set the minimum rectangle width and height allowed in the decomposition. It sets, and is set by, the FhManhGridCnt variable. The minimum size is given by

sqrt(*area_of_interest*/FhManhGridCnt)

The default entry value is 1000. Larger values are more accurate but slow processing, sometimes dramatically. The *area_of_interest* is the layout area being processed for input to *FastHenry*.

FhDefaults
Any text entered into this area will be included in a `.defaults` line in *FastHenry* input. The text must be understood by the *FastHenry* program in use. The text will also be saved in and track the FhDefaults variable.

FhDefNhinc
This entry tracks the value of the FhDefNhinc variable, which sets a default value for the *FastHenry* `nhinc` parameter. This will be overridden by values set by the layer keyword `FH_nhinc` in the technology file, unless the FhOverride variable is set, in which case the variable has precedence.

FhDefRh
This entry tracks the value of the FhDefRh variable, which sets a default value for the *FastHenry* `rh` parameter. This will be overridden by values set by the layer keyword `FH_rh` in the technology file, unless the FhOverride variable is set, in which case the variable has precedence.

Override Layer NHINC, RH
When this check box is set, the values of the **FhDefNhinc** and **FhDefRh** entries override values set with technology file keywords on individual layers. This tracks the state of the FhOverride variable (set or not).

Use FastHenry Internal NHINC, RH
If set, the `>nhinc` and `rh` values are passed to *FastHenry*, which will use the values internally for filament generation. If not set, the values will be used in the input file generation to refine segments according to the same parameters, and *FastHenry* will not create additional filaments. This button tracks the state of the FhUseFilament variable (set or not).

The **FastHenry Volume Element Refinement** allows one to crudely refine the raw segmentation. By default, this is not enabled, so that only tiling provides refinement. This may be a good starting point for a third-party refinement algorithm, but with present *FastHenry* programs is unlikely to provide accurate results as-is. When enabled, the crude refinement should provide somewhat better results.

The refinement algorithm works as follows. For each conductor, find the volume and divide by the film thickness. The maximum size is taken as the square root of the sum of these terms divided by the FhVolElTarget as entered. If a side of a tile exceeds this length, it is subdivided. This is repeated until no tiles have sides larger than the calculated length. The total number of tiles (or "volume elements") is approximately the target value entgered. The total number of segments is approximately six times larger.

Additionally, a minimum dimension can be defined. Volume elements with a dimension smaller than the minimum will be ignored.

The refinement is "crude" due to each refined volume element being approximately the same size. If the size is small enough, sufficient spatial resolution for accurate calculation is achieved. This resolution is needed along edges, and at corners, where there are strong field gradients, but is gross overkill for most areas. Since the solving time is related to the total number of segments, this type of refinement is very inefficient with respect to memory use and execution speed.

Enable

This check box will enable or disable the refinement. When pressed, the two numerical entry areas will become un-grayed, and internally the FhVolElTarget variable will be set to the number shown in the **FhVolElTarget** entry area, and the FfVolElMin variable will be set to the number shown in the **FhVolElMin** entry area. When the **Enable** button is set inactive, the FhVolElTarget and **FhVolElMin** variables are unset. The FhVolEnable variable tracks the state (set or unset) of the **Enable** button.

FhVolElTarget

This entry area is sensitive only when the **Enable** check box is checked. It tracks the value of the FhVolElTarget variable, which can be set to a real value of 1e2 – 1e5. This will be the approximate number of refined tiles generated in the input file.

FhVolElMin

This entry area is sensitive only when the **Enable** check box is checked. It tracks the value of the FhVolElMin variable, which can be set to a real value of 0 – 1.0. Volume elements with a dimension smaller that this fraction of the maximum dimension computed for the volume element target will be ignored.

The Jobs page

The **Jobs** page contains a list of running background jobs. Each entry provides the process identification number (PID), the name of the executing program, and the local date and time when started. Entries can be selected by clicking with the mouse.

When an entry is selected, the **Abort job** button below the list becomes un-grayed. Clicking this button will terminate the selected process. The user should consider that there is no confirmation and no ability to resume the run.

This page intentionally left blank.

Chapter 17

The User Menu: User Commands and *Xic* Scripts

The **User Menu** contains built-in commands listed in the table below.

User Menu			
Label	**Name**	**Pop-up**	**Function**
Debugger	`debug`	**Script Debugger**	Debug scripts
Rehash	`hash`	none	Rebuild **User Menu**
others	—	—	User scripts and menus

Other buttons which appear in the **User Menu** execute user-generated scripts, or pop up menus of user-generated scripts. *Xic* provides a powerful native language, from which the user can automate various tasks. The **User Menu** is the primary means to execute scripts, though the **!exec** command provides a non-graphical alternative.

The default system-wide location for scripts is in the directory `/usr/local/xictools/xic/scripts`, however this can be reset with the XIC_SCR_PATH environment variable, or defined in the technology file with the `ScriptPath` keyword. The syntax is the same as for other *Xic* search paths, for example:

`ScriptPath (` *directory directory1*... `)`

This path can also be set with the ScriptPath variable using the **!set** command. A script path set with the ScriptPath variable takes precedence over a script path defined in the environment using the XIC_SCR_PATH environment variable. If no script path is specified in the technology file, the effective path used will consist of the single default directory.

Each directory in the search path is expected to contain script files, which must have an extension "`.scr`", function libraries which are named "`library`", and script menu files, which will produce a drop-down sub-menu in the **User Menu**. *Xic* provides a library capability which allows code to be shared between scripts. Script menu files must have an extension "`.scm`". In addition, auxiliary files such a images, data, or documentation files may also be present, for use in certain scripts. These will be ignored when searching for scripts.

The default button label in the **User Menu** for a script found in the search path is the base name of the script file, i.e., the file name with the `.scr` stripped off. However, if the first non-blank line of the script file is of the form

`#menulabel` *label*

then the **User Menu** button will use the text in *label*. If the *label* text contains white space, it must be quoted. This text can contain punctuation, though some characters may be stripped or replaced internally. The *label* text must be unique in the top level of the **User Menu**, duplicate entries will not be added.

Scripts can also be included in the technology file itself. These scripts will also appear as buttons in the **User Menu**, as with other scripts. This can be useful for including simple technology-specific commands, such as those that create special extraction layers. However, scripts defined in the technology file can not be loaded into the debugger.

The **!script** command is yet another means by which scripts can be placed into the **User Menu**. This command associates a label, which will appear on the menu button, with an arbitrary path to a script file. Commands registered in this way can also be removed with the **!script** command.

Each command button label in the **User Menu** is unique in the menu or sub-menu where it resides. If a duplicate label is found during the search along the search path, that script will not be added to the menu, and the existing entry will be retained. However, scripts added from the technology file and with the **!script** command are stored somewhat differently, so label text clashes can occur. The following priority is observed in this case.

1. Scripts defined with the **!script** command.
2. Scripts found in the script search path and menus.
3. Scripts found in the technology file.

An encryption capability for scripts is provided. This allows the content of scripts to be hidden from users.

17.1 Example Scripts in User Menu

The *Xic* installation provides some example scripts, which will appear in the **User Menu**. To use these buttons (or any menu buttons) while in help mode, press **Shift** while pressing the menu button.

blackbg: Use a black drawing window background.
This reverts drawing window colors to use a black background, which is the default.

fullcursor: Set/unset full-window cursor.
This command executes a script that toggles whether the FullWinCursor variable is set. When set, the default cursor consists of horizontal and vertical lines that extend completely across the drawing window. The lines intersect at the nearest snap point in the current window.

paths: Set search-path script.
This command executes a script which allows the search path variables to be edited graphically. These variables are otherwise set with the **!set** command, or from the technology file.

spiral: Spiral generator demo script.
This is a text-based command for creating a spiral feature. A series of prompts is given on in the prompt line, where the user supplies dimensions, number of turns, etc. When the prompts are complete, an outline of the spiral is attached to the mouse pointer, and will be instantiated in the drawing window where the user clicks, on the current layer.

spiralform: Form-based spiral generator demo script.
This is a graphical version of the **spiral** script, where the user fills in a form instead of responding to prompts. This is a demonstration of the capability of *Xic* to use HTML forms as a front-end to command scripts.

whitebg: Use a white drawing window background.
Switch to use of a white background color in drawing windows. This adjusts certain attribute colors for greater contrast, and is intended for schematics.

whitebw: Use black on white in drawing windows.
Similar to **whitebg** except that non-background is black. This is intetended for export of schematics to black and white media.

yank: Copy geometry to new cell demo script.
This example script allows the user to copy all geometry in a rectangular area, independent of hierarchy, to a new flat cell. The user clicks twice to define the area, and responds to the prompt for a new cell name. All geometry in the area is copied, clipped to the area, and added to the new cell. The original objects are not affected.

17.2 Script Menus: User-Defined Sub-Menus

Sub-menus in the **User Menu** are produced by a type of library file, "script menus", which (at the top level) are found in the directories in the script search path. The script menus *must* have an extension "`.scm`" ("script menu"). The format is similar to library files:

```
(Library libname);
# any comments

# optional keywords to implement conditional flow
Define [eval] name [value]
If expression
IfDef name
IfnDef name
Else
Endif

[nosort]
name1 path_to_script
...
[name2] path_to_menu
...
```

The first line must be a CIF comment line in the same format as other library files. The *libname* contains the text which will appear in the menu button which will pop up the menu. This text may contain white space and/or punctuation, though some special characters, such as '/', may be stripped or replaced internally. The text can be quoted, though this is optional. The text can also not appear at all, in which case the label used will be the base name (the file name, stripped of the `.scm` extension) of the menu file.

Blank lines and lines starting with '`#`' are ignored. If a line containing the single word "`nosort`" is found, then the menu entries will be in the same order as in the file, otherwise they will be alphabetically sorted. The **User Menu** itself is always sorted.

All library files (including the device library) support a limited macro capability. The macro capability makes use of the generic macro preprocessor provided in *Xic*, which is described in 18.1. The reader should refer to this section for a full description of the preprocessor capabilities. The preprocessor provides a few predefined macros used for testing (and customizing for) release number, operating system, etc. The keyword names, which correspond to the generic names as described for the macro preprocessor, are case-insensitive and listed in the following table.

Keyword	**Function**
`Define`	Define a macro.
`If`	Conditional evaluated test.
`IfDef`	Conditional definition test.
`IfnDef`	Conditional non-definition test.
`Else`	Conditional else clause.
`Endif`	Conditional end clause.

These can be used to conditionally determine which parts of the file are actually loaded when the library is read. The paths (but not the names) are macro expanded, and the conditional keywords can be used to implement flow control as the file is read. They work the same as similar keywords in the technology file (see A.2) and in scripts (see 18.8), and are reminiscent of the preprocessor directives in the C/C++ programming language.

The `Define eval` construct can access functions found in a script library file (see 17.3) found in the same script search path component directory as the menu file file, or from library files found earlier in the search path. When traversing the script search path, the library file, if any, is loaded before the script files and menu files are read.

The remaining lines in the file are name/path pairs, where the *name* is the label that will appear on the button in the pop-up menu, and the *path* is a *full* path to a script file (with "`.scr`" extension) or another script menu file (with "`.scm`" extension) for a sub-menu. If the path is to a menu file, the pop-up menu will contain a button which will produce another pop-up menu containing the referenced menu file's entries. There is no limit on the depth of the references. In this case, the *name* can be omitted, in which case the referenced menu file will supply the button text. If a *name* is given, it will supersede the button text defined in the referenced menu file.

A *name* must always be given for a path to a script file. If the label text in *name* contains white space, it must be quoted. Punctuation is allowed, though some characters may be stripped or replaced internally. Each *name* text should be unique in the menu, duplicates are ignored.

Scripts referenced through a menu file should not be kept in the script search path directories, as they would be added to the main **User Menu** as well as the pop-up menu. They can be placed, for example, in a subdirectory of the directory containing the menu file, which is not itself in the script path.

Only scripts which are defined in separate files can be referenced through a script library, not those defined in the technology file. Scripts defined in the technology file, and those added with the **!script** command, will appear in the main **User Menu**.

Example:

Suppose that you have a `submenu.scm` file, and you want to be able to set the command paths at program startup, depending on some factors. One way to do this is to white a function and place it in the script `library` file, that will return a path to a directory containing the menu functions, e.g.,

```
function func_loc()
  if (something)
    return ("/home/bob/commands")
  else
    return ("/home/joe/commands")
  end
endfunc
```

In the `submenu.scm` file, one has lines like

```
define eval FUNC_LOC func_loc()
cmd1 FUNC_LOC/cmd1.scr
cmd2 FUNC_LOC/cmd2.scr
...
```

In this example, the menu appearance is always the same, however the functions executed when a button is pressed depend on the `func_loc()` return.

17.3 Script Libraries: Code Sharing

Scripts are executed in *Xic* using a high-performance compilation technique whereby the entire script is first compiled, then executed. Looping constructs within the script execute very quickly. Further, scripts can call user-defined functions that have been saved in a library, avoiding the tiny compilation overhead and allowing the user to build a collection of sharable function blocks.

Files named "`library`" in the script search path are read and processed when *Xic* starts, and during a **Rehash** command. These files should contain function definitions. The functions will be "compiled" and saved within *Xic*. Any executable lines that are not part of a function block will be executed once only as the library is read. This can provide initialization, if needed.

Functions that are saved will be available for calling from scripts, avoiding having to parse them each time the script is run. This also facilitates using the same functions in several scripts.

The functions saved within *Xic* can be maintained with two '!' commands: **!listfuncs** provides a pop-up listing of the functions stored, and **!rmfunc** allows the user to remove functions from memory.

17.4 Encrypted Scripts

Script encryption allows script files to be encoded so as to be unreadable without a password. This allows OEMs to provide script packages to users while maintaining confidentiality of the script content.

The encryption method is strong enough to foil most attempts at breaking the code by average users, however it is probably easily broken by experts. The encryption algorithm is not export-restricted.

Encryption and decryption of script files is implemented with two utilities, which are provided in the Accessories distribution. Also provided with the accessories is a utility for changing the default password compiled into the *Xic* executable. There is also a related script function, and a related command-line argument to *Xic*.

The encryption/decryption utilities are:

`wrencode` *file* [*files* ...]
`wrdecode` *file* [*files* ...]

Both programs take as arguments lists of files to encode or decode. At least one file must be specified.

The `wrencode` program will prompt the user for a password, and for a repetition of the password. The files on the command line will be encrypted using this password.
WARNING: since the encryption is done in-place, be sure to save a non-encrypted backup of the files.

The `wrdecode` program will prompt once for a password, and will decrypt the files listed in the command line which have been encrypted with this password. They are not touched otherwise.

The encryption/decryption should be portable between all systems that can run these two utilities.

Xic will read plain-text and encrypted scripts. Encrypted scripts can be read only if *Xic* has the correct password, i.e., the one used in the `wrencode` utility to encrypt the scripts. At present, *Xic* can only retain one password at a time.

Xic has a built-in default password, which is active if no other password is specified. This is built into the *Xic* executable file (in encrypted form) and can be changed with the `wrsetpass` utility. The "factory" default password is:

Default password: `qwerty`

The password can be given to *Xic* on the command line with the `-K` option:

`-K`*password*

Note that there is no space between the "`-K`" and the password. As the password can contain almost any character, if the password contains characters which could be misinterpreted by the shell, the password should be quoted, e.g., `-K'`*password*`'`. The password set with the `-K` option overrides the default password.

If the `.xicinit` or `.xicstart` file, or the function library file, or a script run from batch mode, is encrypted, the encryption password must be given to *Xic* with the `-K` option, or be the default password. As the password can be changed with the `SetKey` script function, **User Menu** scripts can in principle use different passwords, which must be set before the script is executed.

It is possible the change the password when *Xic* is running with the `SetKey` script function:

(int) `SetKey`(*password*)

This function sets the key used by *Xic* to decrypt encrypted scripts. The password must be the same as that used to encrypt the scripts. This function returns 1 on success, i.e., the key has been set, or 0 on failure, which shouldn't happen as even an empty string is a valid password.

At most one password is active at a time. If the file can not be opened with the current password, *Xic* will behave as if the file was empty.

17.5 The Debug Button: Enter Script Debugger

The **Debugger** button in the **User Menu**, which unlike most of the other commands in this menu is an internal command, brings up a panel which facilitates script development. The panel contains debugging options such as breakpoints, single-stepping, and text editing.

The text window displays the text of the currently loaded script. In editing mode, the verbatim text is shown. When not in editing mode, the text is shifted to the right by two columns, so that the first column can be used to indicate breakpoints and the current line.

The current mode (editing or executing) is switched by the button to the left of the title bar. The label of this button switches between "`Run`" and "`Edit`" to indicate the mode to switch to. In edit mode, the **Execute** menu is not available. In execute mode, the **Edit** menu is not available, and some functions in the **File** menu, such as **New** and **Load**, will switch back to edit mode.

While in editing mode, the text in the window can be edited, using the same keyboard commands as the text editor pop-up. The text is shown as it appears in the buffer, without the first two columns reserved for breakpoint indication as used outside of edit mode.

The following command buttons appear in the **File** menu.

New
: This button will clear the present contents of the text window, allowing a new script to be keyed in. If the present script is modified and not saved, a message will inform the user, and the text will not be cleared. Pressing the **New** button a second time will clear the text, and the previous changes will be lost.

Load
: The **Load** button will prompt for the name of a script file, which will be loaded into the debugger. A full path must be given to the file, if the file is not in the script search path. If, while the load pop-up is active, a script is selected in the **User Menu**, that script name will be loaded into the load dialog text area.

Print
: The **Print** button brings up a control panel for sending the contents of the text window to a printer, or to a file.

Save As
: This button allows the contents of the text window to be saved in a file. The user is prompted for the name of the file, the default being the original file name, if any. A pre-existing file of the same name will be retained with a "`.bak`" extension.

Write CRLF
: This menu item appears only in the Windows version. It controls the line termination format used in files written using **Save As**. The default is to use the archaic Windows two-byte (DOS) termination. If this button is unset, the more modern and efficient Unix-style termination is used. Older Windows programs such as Notepad require two-byte termination. Most newer objects and programs can use either format, as can the *XicTools* programs.

Quit
: The **Quit** button will retire the debug panel, which is the same effect as pressing the **Debugger** button in the **User Menu** a second time. If there is unsaved text, a message will alert the user, and the panel will not be withdrawn. Pressing the **Quit** button a second time will retire the panel without saving changes. The debugger can also be dismissed with the window manager "delete window" function, which has the same effect as the **Quit** button.

The debugger text window serves as a drop receiver. Files can be loaded by dragging from the **File Selection** panel or another drag source, and dropping into the text window of the debugger, or the small “load” dialog window that receives the file name. The file name will be transferred to the load dialog, which will appear if not already present.

If, while in editing mode, the **Ctrl** key is held during the drop, the text will instead be inserted into the document at the insertion point.

The **Edit** menu contains commands specific to editing mode, and is disabled while in execute mode.

Undo This will undo the last modification, progressively. The number of operations that can be undone is limited to 25 in Windows, but is unlimited in Unix/Linux.

Redo This will redo previously undone operations, progressively.

The remaining entries allow copying of selected text to and from other windows. These work with the clipboard provided by the operating system, which is a means of transferring a data item between windows on the desktop (see 3.13.3).

Cut to Clipboard
Delete selected text to the clipboard. The accelerator **Ctrl-x** also performs this operation. This function is not available if the text is read-only.

Copy to Clipboard
Copy selected text to the clipboard. The accelerator **Ctrl-c** also performs this operation. This function is available whether or not the text is read-only.

Paste from Clipboard
Paste the contents of the clipboard into the document at the cursor location. The accelerator **Ctrl-v** also performs this operation. This function is not available if the text is read-only.

Paste Primary (Unix/Linux only)
Paste the contents of the primary selection register into the document at the cursor location. The accelerator **Alt-p** also performs this operation. This function is not available if the text is read-only.

The **Execute** menu contains commands for executing the script in a controlled fashion. Displaying this menu switches to execute mode. The text is shifted to the right by two columns. The first column is used to indicate the next line to execute, and breakpoints.

The current line, which would be executed next, is shown with a colored ‘>’ in the first column. Clicking on this line will cause the line to be executed, and the ‘>’ will advance to the next executable line (the same as the **Step** menu item). Clicking on any other executable line of text in the text window will set a breakpoint, or clear the breakpoint if a breakpoint is already set on that line. A line containing a breakpoint is shown with a ‘B’ in the first column. Execution, initiated with the **Run** button, will pause before the next line containing a breakpoint, after the current line.

Run
The **Run** button will cause lines of the script to be executed until a line containing a breakpoint or the end of the script is reached. Pressing **Ctrl-c** when a drawing window has the focus will cause the script to pause at the next line.

Step
The **Step** button causes the current line to be executed, and the current line pointer will be advanced to the next line.

Step

The **Reset** button will reset the current line to the start of the script.

In addition to the accelerators listed in the **Execute** menu, there are hard-coded accelerators for the menu functions.

t, **Space**	single step
r	run
e, **Backspace**	reset

A problem with the menu accelerators is that they require the **Ctrl** key to be pressed, which may fool scripts that are sensitive to the **Ctrl** key.

Monitor

The **Monitor** button allows variables to be monitored and set.

After the **Monitor** button is pressed, the user is prompted for the names of variables from the *Xic* prompt line. A list of variable names (space separated) is entered. A pop-up window will appear which lists these variables and their present values. If the variable is undefined or not in scope, the value will be "???". The values are updated after each line is executed. If, in response to the prompt for a list of variables, one enters "`all`" or "`*`" or "`.`", all of the variables currently in scope will be monitored.

Variables being displayed in the monitor window can be set to an arbitrary value by clicking on the variable name in the monitor window. The value will be prompted for on the *Xic* prompt line. Only variables that are in scope will accept a value. This feature can be used to alter program operation as the program is being run. Variables will continue to be monitored until the monitor window is dismissed.

The monitor window in the script debugger can handle multi-dimensional arrays. When specifying an array variable, the variable name can be followed by a range specification, enclosed in square brackets, as follows:

$$[rmin{-}rmax, dim2, dim3]$$

This is entirely optional, as are the individual entries. The three comma separated fields correspond to the three dimensions (maximum) of the array. The lowest dimension can be a range, where *rmin* and *rmax* set the range of indices to print or set. The remaining two fields are indices into the higher dimensions. These indices are taken as 0 if not given. One of the range values can be omitted, with the following interpretations:

`[`*rmin*, ...	Use the single index *rmin*.
`[`*rmin*−, ...	Use the range *rmin* to the length of the lowest dimension.
`[`−*rmax*, ...	Use the range 0 − *rmax*.

White space can appear, and the commas are optional, except in the second form above where a comma must follow the '–'.

If the *rmax* value is less than *rmin*, the printing order of the elements is reversed, as is the order of elements accepted when the variable is being set.

A similar range specification can be applied to string variables. In this case, only the first field is relevant, and the range applies to character positions.

The following commands are found in the **Options** menu of the editor. These commands are always available.

Search

Pop up a dialog which solicits a regular expression to search for in the document. The up and down arrow buttons will perform the search, in the direction of the arrows. If the **No Case** button is active, case will be ignored in the search. The next matching text in the document will be highlighted. If there is no match, "not found" will be displayed in the message area of the pop-up.

The search starts at the current text insertion point (the location of the I-beam cursor). This may not be visible in execute mode, but can be set by clicking with button 1 (which may set a breakpoint, so you will have to click again to remove it). The search does not wrap.

Font

This brings up a tool for selecting the font to use in the text window. Selecting a font will change the present font, and will set the default fixed-pitch font used in pop-up text windows.

17.6 The Rehash Button: Rebuild User menu

The **Rehash** button in the **User Menu** will rebuild the **User Menu**, taking script and menu files found along the script search path and creating the corresponding entries in the **User Menu**. This command should be executed if a new script is added to the path. It is implicitly executed whenever the script path is changed. This function will also load the contents of files named "`library`" found in the script search path. These files contain function definitions only. Like the **Debugger** button but unlike other buttons in the **User Menu**, this is an internal command.

Chapter 18

The *Xic* Scripting Language

18.1 The Macro Preprocessor

As part of the scripting language support, a macro preprocessor package is provided, which is used by *Xic* when reading various types of input. This input includes scripts, library and menu ("`.scm`") files, and the technology file. This section describes the common features of this macro processing system.

A macro is a text token that usually references another piece of text. When lines of text are "macro expanded", the tokens that are recognized as macro names are removed, and replaced by the text associated with the macro name. This is done recursively, as the replacement text may itself contain macro names.

In other cases, macros can be used to identify blocks of text to be discarded when a file is being read. The macro system applies conditional testing based on the existence of a defined macro name, or whether a macro name is set to a certain value, and marks blocks of text for inclusion or exclusion accordingly.

This section will describe the common functionality of the macro preprocessor, and will be referred to in the sections describing the format of the various types of input. Not all features are used in all cases, and the exact keyword names (but not the functionality) will vary for different input types. For example, the keyword which defines a macro is "`#define`" in scripts, but "`Define`" may be used in other types of file.

18.1.1 Predefined Macros

The macro preprocessor defines several macro names that are common to all instances of the preprocessor and apply in all cases where the preprocessor is in use. These names are the same in all cases, they do not differ with different file types. The predefined macro names can not be undefined or set to a different value, attempts to do so will trigger an error. These are the following:

`RELEASE`

First implemented: release 3.0.5
The macro name `RELEASE` is predefined to the release number code. The release number code is a five digit integer *xyzz*0, corresponding to release *x.y.z*. The *x* (always 3) and *y* are one digit fields, *zz* is a two-digit field, 0 padded. The trailing 0 is a historical anachronism. For example, for release

3.2.5, the macro is predefined to "`32050`".

`GENERATION`
First implemented: release 4.1.10
This is set to the generation part of the release triplet, which is "4" for the current generation 4.

`MAJOR`
First implemented: release 4.1.10
This is set to the middle number of the release triplet, for example for release 4.1.10, `MAJOR` is set to "1".

`MINOR`
First implemented: release 4.1.10
This is set to the rightmost number or the release triplet, for example for release 4.1.10, `MINOR` is set to "10".

`OSNAME`
First implemented: release 4.2.12
This is set to the distribution name of the program, for example "`LinuxCentos7`".

`OSTYPE`
First implemented: release 3.2.19
This macro name is set to one of the following words, depending on the operating system target of the running program. Note that this is determined at compile time, so is static in the program binary, and may not be the "real" operating system if running under an emulator. For example, a Linux binary running under FreeBSD would still indicate "`Linux`".

Distribution Target	**Keyword**
Any Linux	"`Linux`"
Windows	"`Windows`"
FreeBSD	"`UNIX`"
Any Apple	"`OSX`"

`OSBITS`
First implemented: release 3.2.19
This macro is set to either "`32`" or "`64`", depending on whether the program was compiled for 32- or 64-bit memory addresses. This is determined at compile time, so that a 32-bit binary running on a 64-bit operating system would indicate "`32`".

`XTROOT`
First implemented: release 3.2.19
This macro is defined to be the system xictools installation location path as assumed by the running program. It reflects the status of environment variables or other means of defining this path, and will revert to a default. This directory is typically "`/usr/local/xictools`" in non-Windows programs. The Windows path is similar but may include a drive specifier and use back instead of forward slash separators.

`PROGROOT`
First implemented: release 3.3.1
This macro is defined to be the system installation location path for the running program as assumed by the running program. It reflects the status of environment variables or other means of defining this path, and will revert to a default. For example, this directory is typically "`/usr/local/xictools/xic`" for the *Xic* program, in non-Windows programs. The Windows path is similar but may include a drive specifier and use back instead of forward slash separators.

product name

First implemented: release 3.0.5

In releases prior to 4.0.9, exactly one of the macro names "`Xic`", "`XicII`", or "`Xiv`" would be defined, depending upon which of the programs was being run. The name is not defined to any text, but one can test whether or not a given name is defined. In release 4.0.9 and later, the separate *XicII* and *Xiv* programs were discontinued, but the functionality lives on as feature sets of *Xic*. The `Xic` symbol is always defined when running *Xic* for any feature set, and is therefor rather useless but provides some backward compatibility.

This macro has the property that instances of the macro are not replaced (with an empty string) when macro-expanding, i.e., macro substitution is inhibited (4.2.12 and later).

feature set name

First implemented: release 4.0.9

The macro "`FEATURESET`" will be defined to one of three strings, depending upon the feature set running. If all features are enabled, the string is `"FULL"`. If the *XicII* (EDITOR) permission set is running, the string is `"EDITOR"`. If the *Xiv* (VIEWER) feature set is running, the string is `"VIEWER"`. The macro can be tested with forms similar to

```
If FEATURESET == "FULL"
...
Endif
```

technology name

First implemented: release 3.2.18

If the technology file uses the `Technology` keyword to define a name for the technology, that name will be predefined as a macro name. The name is not defined to any text, but one may test whether or not a given name is defined.

This macro has the property that instances of the macro are not replaced (with an empty string) when macro-expanding, i.e., macro substitution is inhibited (4.2.12 and later).

These macros are always available, and additional predefined macros may be available in the various contexts, which are documented elsewhere.

technology definitions

First implemented: release 4.3.10

In addition to above, if the technology file uses the `Technology` keyword to define a name for the technology, the predefined macro "`TECHNOLOGY`" is set to that name. Furthermore, if the `Vendoor` keyword is used to define a name, the "`VENDOR`" predefined macro is set to the name. Similarly, `Process` can be used to assign a name to the predefined "`PROCESS`" macro.

18.1.2 Generic Macro Keywords

The following keywords may vary between different contexts where the macro processor is used. The actual keywords are programmable within the macro preprocessor system, so as to better match the syntax of the file format to which the preprocessor is being applied. Here, we will use italicized generic names for these keywords, but the correspondence to actual keyword names (given in the documentation for the specific file formats) should be obvious. The square brackets indicate "optional".

DEFINE [`eval`] *token*

DEFINE [`eval`] *token*(*arg*, *arg*1, ..., *arg*n) [*text_containing_args*]

The macro name *token* may use alphanumeric characters and underscores, and must start with an alpha or underscore character. The name is optionally immediately followed by an argument list in parentheses. The arguments are arbitrary alphanumeric plus underscore tokens that start with an alpha or underscore and are separated by commas. This is the same syntax used in the C language preprocessor for `#define` lines. The remainder of the line is the substitution string.

If the optional "`eval`" keyword is not included, the replacement text, if any, will replace the macro in lines of text being macro expanded.

If "`eval`" is included (this is verbatim but case-insensitive), the replacement text is assumed to be executable as a single line script. The script will be executed, and the result (or return value) will be converted to a text string (if necessary) and taken as the replacement text.

IF expression
The *expression* is a constant expression which can contain macros previously defined with *DEFINE*, predefines, and functions from the script library files or otherwise available in memory. The *expression* is evaluated numerically, and if the result is nonzero (as an integer), the block that follows until the corresponding *ELSE* or *ENDIF* is read. If the result is 0 (as an integer), the block of lines that follow is skipped.

IFDEF token
If *token* has been defined, either with *DEFINE* or as a predefined macro, reading resumes at the following line. Otherwise, reading resumes at the line following the next *ELSE* or *ENDIF*.

IFNDEF token
If *token* has not been defined, reading resumes at the following line. Otherwise, reading resumes at the line following the next *ELSE* or *ENDIF*.

ELSE Used in conjunction with *IF*, *IFDEF* and *IFNDEF*.

ENDIF
Used to terminate an *IF*, *IFDEF*, *IFNDEF*, or *ELSE* block.

In various contexts, other special keywords may be recognized. These are described elsewhere.

Examples:

The examples below illustrate some simple constructs that improve portability of input files, using the predefined macros and generic keywords. In real input, the actual keywords appropriate for the type of file should be used.

The *IF* keyword, and product name and `RELEASE` predefines, were implemented in release 3.0.5, so use is not compatible with older releases. Nevertheless, files can be made portably version dependent through use of *IFDEF* and/or *IFNDEF*.

IFNDEF `RELEASE`
\# old release
text...
ELSE
IF `RELEASE == 30050`
\# release xic-3.0.5
text...
ELSE
\# a later release
text...
ENDIF

ENDIF

Often, it is necessary to know what operating system is being used. Usually, there are really only two categories: Windows, and everything else.

IF `OSTYPE == "Windows"`
running Windows
text...
ELSE
# not running Windows
text...
ENDIF

It may be necessary to disable certain setup if not running the full *Xic* feature set, for example, if the same file is used for different *Xic* feature sets.

IF `FEATURESET == "FULL"`
running Xic
text...
ELSE
IF `FEATURESET == "EDITOR"`
running XicII
text...
ELSE
IF `FEATURESET == "VIEWER"`
running Xiv
text...
ELSE
# impossible!
ENDIF
ENDIF
ENDIF

18.2 Introduction to *Xic* Scripts

Xic supports a scripting language and user-definable commands (scripts). These commands can be associated with buttons in the **User Menu**. Scripts may also be used in "script labels", which are labels placed in a drawing which execute the script when clicked on. Scripts are also used in user-defined design rules, and are the basis for the protocol used in the *Xic* server mode. Scripts are also integral to the native parameterized cell capability. A library of several hundred built-in functions callable from scripts provide control over virtually all of the program capabilities.

In addition to the native scripting capability, *Xic* provides a plug-in interface to the popular open-source Python and Tcl/Tk scripting languages.

The scripting capability can be used to provide commands that quickly generate complex geometry for microwave integrated circuits, for example. Another application is to produce simple, often-needed geometry such as vias or device structures. This powerful capability provides the user with the tools to automate many tasks.

Script files are created using a text editor, perhaps most conveniently from within the debugger built into *Xic*, which is accessible from the **Debugger** button in the **User Menu**. Scripts can be executed within the debugging environment, which offers single stepping, breakpoints, and other features. The language is rather generic and somewhat reminiscent of the C programming language.

18.3 The Scripting Language

A script consists of command lines, each containing one or more syntactically complete statements. Lines may be continued by adding a backslash character at the end of the line, which "hides" the return character. Parentheses are used as delimiters to enforce execution order, and to enclose arguments to functions. Arrays of up to three dimensions are supported, with the array indices separated by commas and enclosed in square brackets. Array names are taken as addresses, and may be passed to functions, and used in arithmetic expressions. There are no address or pointer operators, however a pointer mechanism does exist.

If a line begins with the pound sign '`#`' the line will be ignored by the parser, unless the line contains a "preprocessor" directive, described in 18.8. Preprocessor directives can be used to comment out blocks of lines. The character sequence '`//`' at the start of a line also indicates a comment.

There is one "special case" comment, which must be the first non-blank line of a script file to have relevance:

`#menulabel` *label*

The *label* is a word or quoted phrase, which will appear on the button in the top level of the *User Menu* which executes the script. Otherwise, like any comment, the line is ignored.

Each line of a script generally contains one statement or clause, the entirety of which should be contained in the same logical line. Physical lines can be continued with a backslash character to form a single logical line. If the last character on a line is the backslash (`"\"`) character, the line that follows will be logically appended, replacing the backslash.

The parser will parse the opening clause of a line, and if there is additional text, the parser will continue reading, until all text on the line has been processed. Thus, a single line can actually contain multiple statements. Each statement can be terminated with a semicolon (`";"`) to explicitly terminate the statement. Almost always, this is optional, however there may be rare cases where explicit termination is needed to force the parser into a correct interpretation. The end-of-line will also act as a statement terminator, which is why a statement must appear in a single logical line.

With a couple of exceptions, an entire script can be given on a single line. This is not recommended, as line-numbered error messages would not mean much, and the debugger would be useless, however this facilitates creating complicated macros with the "`#define`" preprocessor directive, which must always expand to a single line.

The two exceptions are:

1. Comments and preprocessor directives start with '#' and continue to the end of the current line. Preprocessor directives must be given at the start of a line, though comments can appear in a line where a new statement could appear. It is not possible to include (unrelated) command text after a comment or preprocessor directive in a line.

2. The declaration lists that follow the `static` and `global` keywords must be terminated with a semicolon if a different construct (including a comment) is to appear on the same line following the `static` or `global` construct.

Scripts can interact with forms in HTML documents so that the form can be used as input for *Xic* scripts. This is often more convenient than issuing a sequence of prompts to the user for input. The forms interface makes use of the HTML viewer used with the help system.

There is an expanding library of internal functions which can be called from scripts, described in F.1.1. The parser also supports user-defined functions.

Identifiers (function and variable names) must start with an alphabetic character or underscore, and can contain digits. Characters other than alphanumerics and underscore are generally not accepted in identifiers and will cause syntax errors. Identifiers are case-sensitive.

18.4 Error Reporting

Compile and run-time error messages go to the standard error channel. That means, in interactive graphical mode, that the messages will appear in the terminal window from which *Xic* was launched. Under Microsoft Windows, if *Xic* is started from an icon or the **Start** menu, a terminal window will be created. This window is usually hidden behind the main graphics window, so one should make this window visible when developing scripts. The same applies to the terminal window under Unix/Linux.

18.5 Data Types

Variables may be one of several different types. The types that are currently implemented are listed below.

no type
: Before a variable receives an assignment, it has no type, but behaves in all respects as a string with a value of the variable name.

string
: The string type contains text data.

scalar
: Scalars are real numbers that are stored internally in double-precision floating point format. Conversion to integer values, such as for array subscription, is performed automatically where needed.

array
: The array type contains a 1–3 dimensional array of numerical values.

complex
: The complex type contains real and imaginary double precision floating-point scalar values. Most math functions and operators accept complex values, and return complex values if passed a complex value.

handle
: The handle type contains a reference to a complex data object. There are a number of different object types that can be referenced by handles.

zoidlist

Zoidlists contain a list of trapezoids that define spatial regions.

layer_expr

This variable type contains a parse tree for a "layer expression" (see 15.1). A layer expression is a logical expression involving layer names.

The type of a variable is determined by its assignment, or in the case of arrays, by declaration. Once a type is assigned, it is generally an error to assign a different type. Exceptions are the undefining of array pointers (to be discussed), the promotion of scalars to handles when a handle is assigned to a scalar, and use of the `delete` operator to unassign a variable and free its contents.

Variables that are referenced before assignment, or after being operated on by `delete`, behave as strings with a string value set to the variable name. For example, if an unassigned variable is passed to one of the print functions the name of that variable will be printed.

Type identification of a literal is by context. A quoted quantity is always taken to be a string, e.g., `"2.345"` is a string. Quote marks can be included in strings by preceding them with a backslash. A number in integer, floating, or exponential format is always taken as a scalar.

18.5.1 Scalars

Scalar variables do not need to be declared, and are type assigned when an assignment is first made. Any unquoted number representation in integer, floating point, or exponential notation is taken as a scalar constant. Character constants enclosed in single quotes (as in C) are accepted, with the value being the ASCII character code. There is a `ToChar` function which converts ASCII codes to a string representation for printing. Also accepted are hexadecimal integer constants in the form

`0x`*hex_number*

For example, `0x0`, `0x2a`, and `0xffff003b` are all valid constants.

In addition to the standard floating-point formats, numbers can be represented using SPICE multiplier suffixes. These are alphabetic characters and sequences shown in the table below, which appear immediately following a fixed-point number or integer. The suffix is case-insensitive. For example, the following tokens all represent the same number: `1000`, `1e3`, `1k`. Likewise: `0.0001234`, `1.234e-4`, `123.4u`.

suffix	multiplier	name
a	1e-18	atto
f	1e-15	femto
p	1e-12	pico
n	1e-9	nano
u	1e-6	micro
m	1e-3	milli
mil	25.4	mil
k	1e3	kilo
meg	1e6	mega
g	1e9	giga
t	1e12	tera

18.5.2 Strings

String variables do not need to be declared, and are type assigned when an assignment is first made. Double quote marks are used to delimit literal strings, and are strictly necessary if the string contains spaces or other non-alphanumeric characters.

Whenever a string is defined as a literal in a script or from the **Monitor** panel in the **Script Debugger**, it is filtered through a function which converts the following escape codes into the actual character value. The escape codes recognized, from ANSI C Standard X3J11, are

`\a`	bell
`\b`	backspace
`\f`	form-feed
`\n`	new-line
`\r`	carriage return
`\t`	tab
`\v`	vertical tab
`\'`	single quote
`\"`	double quote
`\\`	backslash

In addition, forms like "*num*" are interpreted as an 8-bit character with ASCII value the 1, 2, or 3–digit octal number *num*.

When a subscript is applied to a string, the index applies to the string with escapes substituted, e.g., "`\n`" counts as one character. When a string is printed to the **Monitor** panel, the reverse filtering is performed.

A special case is the null string, which can be produced by many of the interface functions, usually to signal end-of-input or an error. A null string has no storage. Null strings are not accepted by some functions, so return values from these functions should be tested.

For example:

```
retstr = Get(blather)
if (retstr == NULL)     # NULL is an alias for 0
#    the string is null
end
if (retstr == "")
#    the string is empty
end
```

This example above also illustrates the overloading of "`==`" for strings.

The notation can be even simpler:

```
if (retstr)
#    the string is not null (but may be empty)
else
#    the string is null
end
```

The `[]` notation can be used to address individual characters in strings. Also, *string1* = *string2* + *number* is accepted, yielding a string pointing at the *number*'th character of *string2*. However, it is a fatal error if *number* is negative, so it is not possible to point backwards into a string. Also, if the *number* exceeds the text length of the string, a fatal error is generated. A fatal error is an error which will terminate script execution.

Strings are not copied in assignment, so if multiple variables point to the same string, they will all see any modifications to the string. For example:

```
s1 = "a string"
s2 = s1 + 2
Print(s2)        # prints "string"
s1[4] = ' '
Print(s2)        # prints "st ing"
Print(s1)        # prints "a st ing"
```

The `Strdup` function can be used to make an independent copy of an existing string.

18.5.3 Arrays

Xic provides arrays with up to three dimensions. The indices are specified as comma-separated expressions enclosed in square brackets which follow the variable name, as in `x[c,d]` for a two dimensional array. The higher dimensions appear to the right, so that `c` in the example is the "inner" index.

Declaring and Defining Arrays

Arrays must be declared either by initial assignment, or by a line consisting of the array name followed by square-bracketed indices representing the maximum index in each dimension. In each case, the number of comma-separated indices sets the dimensionality of the array. In the initial declaration, the indices must be integers and not expressions. Indices are 0-based.

Examples

```
x[2, 4]
# This defines an array x:  five blocks of three values

x[2, 4] = some_expression
# This likewise defines the array, and additionally sets
# the highest index to the result of an expression
```

Note that the numbers in the declaration are *not* sizes, but maximum values. This is different than C. Once an array has been defined, subsequent use allows expressions as the index values.

Initialization

We have seen that array elements can be initialized individually by assignment. It is also possible to initialize all or part of an array as a block, using the syntax below:

array`[`*index*[,...]`]` = `[`*a*, *b*, *c*, ... `]`

The left side represents a starting address, in the format of an array element reference. The outer square brackets are explicit, the inner square brackets represent optional higher dimension indices and are not explicit. The square brackets on the right side are explicit, and entries are separated by commas and optional white space. One can use backslash-continuation to break a long initializer into multiple physical (but not logical) lines. The values from the right side are placed in the array starting at the indicated address, in the natural order of array scalar access. The array size is expanded when necessary. The line also serves to declare the array.

The *a*, *b*, *c*, ... can be expressions, or most commonly simple numbers.

Example:

```
ary[0] = [1, 2, 3, 4]
```

This declares and creates a size 4 array named `ary`, with components 1, 2, 3, 4. This is equivalent to the lines

```
ary[0] = 1
ary[1] = 2
ary[2] = 3
ary[3] = 4
```

Dynamic Resizing

In an assignment, if an index is given that is “too large”, the array will be reconfigured so that the new data point will be included. The existing data in the array will remain.

Example

```
x[2, 4]
x[3, 0] = 2
# The array is now sized as if declared with "x[3,4]"
```

After the assignment, the maximum index for each dimension will be the larger of the previous index and the assigning index.

When assigning values to an array, dimensional indices that are omitted are taken as zero, though at least one value must be supplied.

Example

```
x[2, 4]
x[1] = 3
# This is equivalent to x[1,0] = 3
```

This treatment of missing indices only applies in assignment, and *not* in general references, as will be seen below.

There is one important restriction on dynamic resizing: arrays that have pointer variables pointing at them can not be resized, and arrays can not be resized through a pointer. Pointers are described below.

The `GetDims` function can be used to obtain the current dimensions of an array.

Pointers

A pointer to an array is a variable which points to the data of an array, and behaves as an array itself but does not contain its own data. Pointers can point to the array itself, or to a sub-array of an array with multiple dimensions, or to an offset into the data of a single dimensional array.

The simplest case is a direct assignment to an array.

```
x[2, 4]
y = x
```

In this case, the data (held in `x`) can be accessed through `y` or `x` equivalently. In this special case, `y` is an alias, and the array can be dynamically resized through `y` or `x`.

A more interesting case is provided through use of the overloaded '+' operator. For example

```
x[2, 4]
y = x + 1
```

In this construct, the offset is into the highest dimension of `x`, and the return value is the sub-array found at this offset. In the example, `y` is a "[2]" which is located at the address of `x[0,1]`, i.e., `y[0] = x[0, 1]`, `y[1] = x[1, 1]`, `y[2] = x[2, 1]`.

If `x` is a single dimensional array, `y` would also be a single dimensional array, but accessing the data through the offset. For example

```
x[32]
y = x + 10
```

Then `y[0] = x[10]`, `y[1] = x[11]`, etc.

In general references, but *not* assignments, supplying a smaller number of dimensions to an array will return a sub-array. For example,

```
x[2, 4]
y = x[1]
```

This is equivalent to "`y = x + 1`", and `y` will point to a "[2]" at the location of `x[0,1]`.

```
x[2,4,5]
y = x[2]
z = x[3,4]
```

The variable `y` is a "[2,4]" located at `x[0,0,2]`. The variable `z` is a "[2]" located at `x[0,3,4]`.

When a pointer is defined, a reference count is incremented in the pointed-to array. When this reference count is nonzero, the array can not be resized through the dynamic resizing mechanism. The pointers to an array must be reassigned or undefined to allow resizing of the array. Pointers can be reassigned simply by changing them to point to a different array. This can be done arbitrarily.

```
x[2, 4]
y[32]
```

```
z = x + 1
# can't resize x here
z = y
# now ok to resize x
```

One can undefine a pointer by setting it to 0. Once this is done, the pointer variable has no type, and can actually be reused as another type of variable. It is *not* an integer unless it is assigned to an integer. The same effect may be obtained by applying the `delete` operator.

```
x[2, 4]
y = x + 1
# can't resize x here
y = 0
# now ok to resize x
Print(y)
# will give "y", y has no type and acts like a string
y = 0
Print(y)
# will give "0", y is now an integer
```

In our initial case,

```
x[2, 4]
y = x
```

where the pointer is simply a reference to the array, `y` is not strictly speaking a pointer, but rather an alias. In particular, this has no limitation on resizing. The array data can be resized through `y` or `x`. Thus, arrays can be resized from within function calls if the reference to the array itself is passed to the function, and not a pointer (with an offset).

18.5.4 Complex

Support for complex numbers is provided via the complex data type. The basic math operators and functions accept complex numbers, possibly intermixed with scalar values, and will produce a complex result when given a complex operand when appropriate. Generally, a complex number can be passed to a function expecting a real number, and the real part of the complex number will be used. Similarly, a scalar passed to a function expecting a complex number will be accepted as a complex value with zero imaginary part.

Presently, functions will not produce a complex result unless passed a complex argument. For example, the `sqrt` function, if passed a negative scalar, will return a scalar zero. If passed a complex number with negative real part and zero imaginary part, the return will be the complex square root value as one would expect.

Complex numbers can be created with the `cmplx` initializer function, which takes as arguments two scalar values that initialize the real and imaginary part. There are special functions that return as scalars the real and imaginary values, magnitude, and phase of a complex operand. The `Print` function and similar will print a complex value as a comma-separated pair of numbers enclosed in parentheses.

18.5.5 Handles

Several of the interface functions return "handles", which are variables which contain a reference to a complex data object. The handles are in turn passed to other functions which operate on the referenced data object. If an active handle is passed to the `Print` family of functions, a string giving the type of handle will be printed.

When done with a handle, it should be closed (with the `Close` function) to free the memory used by the data object. The same effect is obtained by applying the `delete` operator to the handle. When iterating over a list-type of handle, the handle will be closed automatically when iteration is complete.

There are many different types of data object that can be accessed with a handle, some examples being:

string lists
database objects
file descriptors
properties

With a few exceptions, notably the file descriptor, a handle generally points to a list of objects, such as the currently selected objects, that can be iterated through. Once the iteration is complete, the handle is automatically closed, and further references will not reference an object.

See the section on math operators (18.6) for a discussion of the operations available on handles.

The `HandleContent` function can be called on any handle, and will return the number of objects that can be referenced through the handle. Zero is returned when the handle has iterated to completion. This function is useful in loops which contain iterations over handles.

If a handle still contains references but it is no longer needed, the `Close` function should be called on the handle, or the `delete` operator applied to the handle, to free internal resources.

18.5.6 Zoidlists

A "zoidlist" is a list of trapezoids, which represents a set of spatial regions. Like handles, zoidlists are created by certain functions, for use in other functions.

As in layer expressions, the logical operators can be applied to zoidlists, with the result being a new zoidlist representing the geometric result of the operation. Available operations include intersection (and), union (or), inversion, and clipping. See the section on math operators (18.6) for a discussion of the operations available on zoidlists.

There is a current "reference" zoidlist which represents the "background". If not explicitly set (with the `SetZref` function), this is taken as the boundary of the current cell. The reference is used in operations such as inversion and exclusive-or where the size of the background must be assumed. Note that this background can be an arbitrary shape.

In binary operators with zoidlists, if one of the operands is an integer, 0 represents an empty list, and nonzero represents the reference list.

If a zoidlist is given to one of the `Print` family of functions, the coordinates are printed, one trapezoid per line, in order x-lower-left, x-lower-right, y-lower, x-upper-left, x-upper-right, y-upper.

Zoidlists can be assigned from other zoidlists, in which case a copy is made internally. If the assigned-to zoidlist already contained a list, that list is freed from memory.

18.5.7 Lexpers

The layer_expr variable contains a parsed layer expression. A layer expression is an expression consisting of layer names and logical operators. A layer expression is evaluated within a certain region, representing part of a physical layout, and returns the regions where the layer expression is "true".

A layer_expr is a piece of compiled code that can execute very quickly. Functions that accept a layer_expr argument will generally also accept a string containing the layer expression, and will compile the string before use. If an expression is to be used multiple times, if is far more efficient to pass a layer_expr variable.

These variables can not be assigned, and no operators can be applied. They can be passed to functions only.

If passed to the `Print` family of functions, the layer expression string will be printed.

18.6 Math Operators

The following mathematical operations are supported:

Symbol	Arity	Description
+	binary	addition
−	unary	negation
−	binary	subtraction
++	unary	pre- and post-increment
−−	unary	pre- and post-decrement
*	binary	multiplication
/	binary	division
%	binary	remainder, e.g., 5%3 = 2
^	binary	power, x ^ y = x to power y
& , `and`	binary	and, value is 1 if both operands are nonzero
$\vert$, `or`	binary	or, value is 1 if either operand is nonzero
! , ~ , `not`	unary	not, value is 1/0 if operand is zero/nonzero
> , `gt`	binary	greater than, value is 1 if left operand is greater than the right
>= , `ge`	binary	greater or equal, value is 1 if left operand is greater or equal to the right
< , `lt`	binary	less than, value is 1 if the left operand is less than the right
<= , `le`	binary	less or equal, value is 1 if the left operand is less than or equal to the right
!= , `ne` , <> , ><	binary	not equal, value is 1 if the left operand is not equal to the right
== , `eq`	binary	equal, value is 1 if the left operand is equal to the right
=	binary	the left operand takes the value of the right, and the value is that of the right operand. The type of the left operand becomes that of the right.

The operator-equivalent keywords (`gt`, `lt`, `ge`, `le`, `ne`, `eq`, `or`, `and`, `not`) are recognized without case sensitivitty.

A variable type is determined by its first assignment, of by declaration for arrays. It is generally an error to attempt to redefine a variable to a different type, though if a scalar is assigned from a handle, the scalar type is promoted to handle type.

Note that all operators, including assignment, return a value. Thus, expressions like $3*(x > y)$ make sense (the value is 0 or 3). Binary truth is indicated by a nonzero value.

The increment/decrement operators (`++/--`) behave as in the C language. That is

$y = x++$ is equivalent to $y = x; x = x + 1$
$y = x--$ is equivalent to $y = x; x = x - 1$
$y = ++x$ is equivalent to $x = x + 1; y = x$
$y = x--$ is equivalent to $x = x - 1; y = x$

All of these operations apply to scalar or complex values. If complex and scalar values are mixed, scalar operands are promoted to complex with zero imaginary value. If a complex operand is given, the result is also complex, except for comparison and logical operators, and modulus, which always return scalar values. Comparison operators are applied to both real and imaginary parts, and both must separately satisfy the relation. Increment and decrement operations apply only to the real part of a complex value. In logical operations, a complex value is "false" if both the real and imaginary parts are 0.

18.6.1 Operator Overloading

In general, the operators apply only to numerical variables. However, some of these operators can be used with particular variable types, in which case a function, relevant to that variable, is invoked. In most cases, this is equivalent to invoking an actual function call from the user interface. If a non-numeric variable is supplied to an operator for which no overload exists, the script will generally abort with an error.

String Overloads

The operators $==, !=, >, >=, <, <=$ have been overloaded for strings. If the two operands are strings, the C `strcmp` function is invoked to compare the two strings. If either string is null, it is treated as if it has a lexically minimal value. Either operand can be a scalar 0, which is treated as a null string. Thus, forms like `if (string == 0)` can be used to test for a null string. Null strings, which have no storage, are produced be some script functions. These are different from empty strings, produced for example by `string = ""`, which contain an invisible string termination character.

The + operator has been overloaded for strings to perform concatenation, similar to the `Strcat` library function. The expression `s3 = s1 + s2` is equivalent to `s3 = Strcat(s1, s2)`.

The + and − operators can be applied where the first argument is a string and the second argument is a scalar, and vice-versa in the case of +. The result of the operation is a pointer into the string, which behaves as a string with the first character at the offset given by the scalar. An error is generated if the offset is negative, or is beyond the end of the string.

The − operator can be applied where both operands are strings. The result is a scalar variable representing the difference between the memory addresses of the two strings. This is only useful if both operands are references to the same string.

The `!` operator can be applied to strings. The construct is true only if the string variable contains a null string.

Array Overloads

Pointer arithmetic is discussed in the section describing array variables (18.5.3).

Handle Overloads

Handles can be used in conditional and logical expressions using the and (&), or (||), and not (!) operators. If the handle is non-empty, it is "true", otherwise it is "false". This can be used as a far more efficient loop termination test than a call to `HandleContent`.

The relational operators have been overloaded for handles. The behavior for handles is the same as for scalars, with the handle index being used in the comparison. This is not expected to be useful, except perhaps for file descriptor handles.

The + operator is overloaded to perform concatenation, equivalent to a call to the `HandleCat` function. The syntax is

```
[h1 =] h2 + h3
```

This applies only to handles that contain a list of data items. Both `h2` and `h3` must contain lists of the same type of data. The list in `h3` is copied and pasted on the end of `h2`. If a left hand side is given, it will be assigned the `h2` handle value and be equivalent to `h2`. Most of the time, this is not needed.

The increment operator ++ is overloaded to perform iteration, equivalent to a call to `HandleNext` or similar functions. The postfix and prefix forms are equivalent. The return value is simply a copy of the handle, so again use in an assignment is unlikely to be needed often.

Without overloading, code to iterate over a list handle would appear as

```
h = func_returning_list_handle()
while (HandleContent(h) != 0)
    (do something)
    HandleNext(h)
done
```

Making use of overloading, the same loop could take the following form:

```
h = func_returning_list_handle()
while (h)
    (do something)
    h++
done
```

Zoidlist Overloads

The math and logical operators are overloaded for zoidlists as follows:

+, |	union
−	and-not
*, &	intersection
^	exclusive or
!	inverse

The result of the operation is a new zoidlist, with neither of the operands affected.

To test for an empty zoidlist, the `==` and `!=` comparisons to the value 0 can be applied. Note that "`if (!zlist)`" is an incorrect test for an empty zoidlist; it will invert the list and return true if the inverted list is not empty.

There is a current "reference" zoidlist which represents the "background". If not explicitly set (with the `SetZref` function), this is taken as the boundary of the current cell. The reference is used in operations such as inversion and exclusive-or where the size of the background must be assumed. Note that this background can be an arbitrary shape. In binary operators with zoidlists, if one of the operands is a scalar, 0 represents an empty list, and nonzero represents the reference list.

18.7 Control Structures

As in C, logical "true" is indicated by a nonzero value. The following control statements are accepted.

18.7.1 `delete`

`delete` [*variable*]

Although not strictly a "control" keyword, the `delete` operator is handled at the control-block level. The operator will return the variable to its undefined state, as if before any assignment, and free the contents. After the `delete` operator is applied, the variable can be assigned to any data type.

Using the `delete` operator on an array will remove the array characteristics, so in general the variable can not subsequently be used as an array, except by assigning the variable to another array. The `FreeArray` function can be used to clear the data while still preserving the variable as an array, so that values can still be directly assigned at indices.

The delete operator applied to a handle will close the handle, as if the `Close` function was called. However, the handle will become an undefined variable after `delete`, rather than a scalar 0.

There are two reasons why this operator exists. The user may wish to delete unused variables that contain large data blocks to conserve memory. Also, the `ConvertReply` function can return a variable of any type, thus we must have an undefined variable to take the return, which is impossible in a loop without use of the `delete` operator.

The `delete` operator will generally delete the contents, however for arrays and strings, if the variable has an alias, the content will be retained in the alias, and all pointers or substrings remain valid. If the array or string variable has no alias, any associated pointers or substrings will also be reinitialized, and the underlying data will be freed from memory. Deleting a pointer or substring variable causes that variable to be undefined, but does not affect the pointed-to data.

18.7.2 `return`

`return` [*expression*]

If the `return` keyword is encountered in the main part of a script, execution of the script terminates at that point, and the value returned from any following *expression* is saved. This return value is available as a return from the `Exec` function, if that command was used to execute the script. In general, the return value is ignored.

If used in a function (see 18.10), the function returns immediately with the value of the *expression*, if given.

18.7.3 `if, elif, else`

```
if expression1
    statements1
elif expression2
    statements2
...
elif expressionN
    statementsN
else
    statements
end
```

If *expression1* evaluates nonzero, *statements1* will be executed, otherwise if *expression2* evaluates nonzero, *statements2* will be executed, and so on. If none of the expressions evaluate nonzero, *statements* following `else` will be executed. The only parts that are mandatory are `if`, *expression1*, and `end`, all other clauses are optional.

Note that `elif` is *not* the same as "`else if`". The following two blocks are equivalent:

```
# example using "elif"
if (a == 1)
    Print(1)
elif (a == 2)
    Print(2)
else
    Print("?");
end

# example using "else if"
if (a == 1)
    Print(1)
else
    if (a == 2)
        Print(2)
    else
        Print("?")
    end
end
```

In particular, a common error is the following:

```
if (a == 1)
    Print(1)
```

```
else if (a == 2)
    Print(2)
else
    Print("?")
end
```

This is missing an "`end`" statement (see the second form above).

18.7.4 ternary conditional

a ? *b* : *c*

The ternary conditional operation, familiar from the C programming language, is supported. In this construct, the '`?`' and '`:`' are literal, the *a*, *b*, and *c* are expressions. If *a* evaluates as `true`, then *b* is evaluated and the construct returns its result. Otherwise, *c* is evaluated and the construct returns that result. Hence, the form

```
x = a ?  b :  c
```

is equivalent to

```
if (a)
    x = b
else
    x = c
end
```

The "`true`" condition depends on the type of variable represented by *a*, as for the `if` operator. For example, the following are `true`:

- A nonzero numeric value. This includes the result of a conditional expression when the condition is satisfied.
- An active handle.
- A non-empty zoidlist (a layer expression is evaluated to obtain a zoidlist).
- A non-null string.

18.7.5 `repeat`

`repeat` *expression*
    statements
`end`

Execute *statements* *n* times, where *n* is the integer result of evaluating *expression*. The *expression* is evaluated once only when the block is entered, and the integer value computed is used as the loop counter. The value is tested for zero, which will terminate the loop, and is decremented after each pass. A negative value will produce an error and the script will terminate.

18.7.6 `while`

`while` *expression*
    statements
`end`

On each pass through the loop, if *expression* evaluates nonzero, execute *statements*, otherwise exit the loop.

18.7.7 `dowhile`

`dowhile` *expression*
    statements
`end`

On each pass through the loop, execute *statements*, then evaluate *expression*. If *expression* evaluates to zero, exit the loop.

18.7.8 `break`

`break` [n]

In a loop, the `break` statement will exit the loop. If an integer n is given, control reaches the bottom of the n'th enclosing loop.

Example:

```
while x <= 100
  while y <= 50
    while z <= 20
       statements
       if (x + y + z == 10)
           break 2
       end
    end
  end
# break will jump here
end
```

18.7.9 `continue`

`continue` [n]

In a loop, `continue` causes the loop to be reentered from the top. If an integer n is given, the n'th enclosing loop is reentered.

Example:

```
while x <= 100
  # continue will jump here
  while y <= 50
    while z <= 20
       statements
       if (x + y + z == 10)
           continue 2
       end
    end
  end
end
```

18.7.10 goto, label

`goto` *name*
`label` *name*

Execution can jump to an arbitrary location in a routine with the `goto` statement. Execution resumes at the statement following the associated `label`.

Example:

statements
```
if (z != 0)
    goto error
end
```
statements ...
```
label error
Print("error occurred")
```

18.8 "Preprocessor" Directives

The script parser interprets C-like "preprocessor" keywords. Unlike C, there is only a single pass through the text, so "preprocessor" is a misnomer.

The script preprocessor utilizes the generic macro preprocessor (see 18.1) used in various places within *Xic*. In the present context, the keywords start with the comment '#' character.

In addition to the predefined macros of the generic macro preprocessor, the following predefined macro is used in scripts.

`THIS_SCRIPT`
For any script which is read from a file (not counting the technology file) the token `THIS_SCRIPT` is effectively defined to be the name of the script (for scripts launched from the **User Menu**) or a path to the file. Thus, in the script, the token `THIS_SCRIPT` is replaced by the file or script name. For example, to print the script name in the console window, one could add a line

```
Print("The name of this script is THIS_SCRIPT")
```

The following "preprocessor" keywords are understood in scripts. These pretty much follow the C/C++ standards and behave similarly, and correspond to the generalized keywords described for the macro preprocessor. These are:

Keyword	**Function**
`#define`	Define a macro.
`#if`	Conditional evaluated test.
`#ifdef`	Conditional definition test.
`#ifndef`	Conditional non-definition test.
`#else`	Conditional else clause.
`#endif`	Conditional end clause.

In addition, the following keyword, which has no counterpart in the generic macro preprocessor, is recognized in scripts:

`#macro`
The `#macro` directive, which has no counterpart in C, is assumed to be followed by macro statements in the format used in the `.xicmacros` file, followed by `#end` or `#endif`. If the `#macro` sequence appears in a script file, the macro is defined at that point.

Throughout the script, each line is macro expanded. The actual arguments replace the formal arguments (if any) in the substitution text, which replaces the macro reference. The macro is recognized as a text token, i.e., it must be surrounded by punctuation or white space.

18.9 Math Functions

The following functions apply to complex numbers.

Name	Returns
`cmplx(`x`, `i`)`	Return a complex number
`real(`c`)`	Return real part of a complex number
`imag(`c`)`	Return imaginary part of a complex number
`mag(`c`)`	Return the magnitude of a complex number
`ang(`c`)`	Return the phase of a complex number

The `cmplx` function is used to initialize a complex number through assignment, for example

```
cx = cmplx(1.0, 0.5)
```

creates a complex number `cx` with value `(1.0 + j0.5)`. The other functions take a complex number as an argument, and return a scalar result.

The following math functions are defined internally, and all take scalar or complex arguments.

Name	Returns
`abs`(x)	absolute value or magnitude of x
`acos`(x)	arc-cosine of x
`acosh`(x)	arc-hyperbolic cosine of x
`asin`(x)	arc-sine of x
`asinh`(x)	arc-hyperbolic sine of x
`atan`(x)	arc-tangent of x
`atan2`(x, `y`)	arc tangent of x, y
`atanh`(x)	arc-hyperbolic tangent of x
`cbrt`(x)	cube root of x
`ceil`(x)	smallest integer $>=$ x
`cos`(x)	cosine of x
`cosh`(x)	hyperbolic cosine of x
`erf`(x)	error function of x
`erfc`(x)	complementary error function of x
`exp`(x)	e to the x power
`floor`(x)	largest integer $<=$ x
`gauss()`	gaussian random number
`int`(x)	truncated integer value of x
`j0`(x)	Bessel function order 0 of x
`j1`(x)	Bessel function order 1 of x
`jn`(x, n)	Bessel function order n of x
`ln`(x)	natural logarithm of x
`log`(x)	natural logarithm of x, see below
`log10`(x)	base 10 logarithm of x
`max`(x, y)	largest of x, y
`min`(x, y)	smallest of x, y
`pow`(x, y)	x to the y power
`random()`	random value in $[0, 1)$
`rint`(x)	integer nearest to x
`seed`(x)	seed random number generator
`sgn`(x)	$+1, 0, -1$ if $x > 0, x = 0, x < 0$
`sin`(x)	sine of x
`sinh`(x)	hyperbolic sine of x
`sqrt`(x)	square root of x
`tan`(x)	tangent of x
`tanh`(x)	hyperbolic tangent of x
`y0`(x)	Neumann function order 0 of x
`y1`(x)	Neumann function order 1 of x
`yn`(x, n)	Neumann function order n of x

Most of these functions, when given complex arguments, will compute a complex result. The `atan2`, `seed`, Bessel, Neuman, and error functions ignore any imaginary parts and compute a real value only. The `ceil`, `floor`, `int`, `rint`, and `sgn` functions apply the operation to real and imaginary parts of complex arguments. The `min` and `max` functions generate a complex result containing the operations applied to the real and imaginary parts of the arguments, if at least one argument is complex. The functions listed that take no arguments return scalar values.

With scalar arguments, these functions behave as the corresponding functions in the C library, though the random number functions are specialized to *Xic*. The `seed` function applies a seed value to the random number generators. This can be used to ensure that successive runs using random numbers choose different values. The seed value given is converted to an integer before use. The `random` function returns a random value in the range $[0--1)$. The numbers generated have a uniform distribution. The `gauss` function returns Gaussian random numbers with zero mean and unit deviation.

Note regarding the log function
In *Xic* releases prior to 3.2.23, the `log` function returned the base-10 logarithm. This definition was changed in 3.2.23, and the `log10` function added, for consistency with programming languages, *WRspice*, and most other software. This will require users to update legacy scripts that use the `log` function to call `log10` instead. However, there is a `LogIsLog10` variable that can be set to revert `log` to base-10. This can be used temporarily, but is not recommended for the long-term.

18.10 User-Defined Functions

In scripts, user-defined functions are supported. The function must be defined before it is called.

A function definition starts with the keyword `function`, followed by the function name and argument list. The keyword `endfunc` terminates the definition. These blocks can appear anywhere between statements in a script file, however they must appear before any calls to the function. Once a function has been parsed, it is added to an internal database, where a compiled representation is retained. If the same function is parsed again, the in-memory representation is updated. There is a mechanism for automatically loading libraries (see 18.2) of script functions at program startup. Use of this mechanism avoids the overhead of repeatedly parsing function definitions that are found in script files.

The function is called just like a built-in function. Scalar variables are passed by value, other types are passed to the function by reference. Variables defined within a function are automatic by default.

Functions can return a value. In a function, the construct

`return` [*expression*]

can be used to terminate execution, and the value of the *expression* is returned by the function. The value returned can be of any type. If the return value is a local string, the string will be copied. If the return value is a pointer to an array, the array must have been passed as an argument or have been declared `static` or `global` (see 18.12).

The example below illustrates how variables are passed, and the scope for changes. Strings and arrays can not be redefined in a function, but elements can be changed. Arrays are used to pass results back to the calling function.

Example:

```
function myfunc(a, b, c)
Print(a, b, c)
endfunc

function examp(a, b, c)
# a is a constant, can be redefined within the scope of examp
a = 2
# b is a string, it is an error to redefine, but can be altered, in
# which case the string is altered in the calling function as well
#b = "b string"   (this produces an error)
```

```
b[2] = 'x'
# c is an array, it is an error to redefine, but elements can be
# changed, in which case this is reflected to all users of the array
# c[3]  (redefinition, this is an error)
c[1] = 1.234
myfunc(a, b, c)
endfunc

Print("this is a test")
x = 1
y = "a string"
z[2]
myfunc(x, y, z)
examp(x, y, z)
myfunc(x, y, z)
```

It is presently not possible to single-step through a function in the **Script Debugger**.

18.11 The exec Keyword — Immediate Execution

Script execution is a two-step process: first, the text of the script is parsed, and executable data structures are created internally, and second, the execution is performed. Consider the following script:

```
Set("ScriptPath", "/path/to/library_dir")
some_library_function()
```

Naively, the first line will set the script path to the directory containing the `library` (see 18.2) file, and the second line will execute a function from the library. However, this will not run, since the library function must be resolved before the parser can process the function call. Somehow, we must ensure that the `Set` line is executed before the following line is parsed.

The `exec` keyword will perform this trick. When an `exec` keyword is encountered, the remainder of the line (or to the next semicolon) is parsed and executed immediately, and is *not* added to the parse tree for scheduled execution with the other lines. Thus, the example above should be

```
exec Set("ScriptPath", "/path/to/library_dir")
some_library_function()
```

Multiple `exec` lines are executed in order of appearance. Variables can be used and set, but remember that this will be done before any manipulation from the normal script lines. For example, the ScriptPath switch can be hidden:

```
exec tmppath = GetPath("ScriptPath")
exec Set("ScriptPath", "/path/to/library_dir")
some_library_function()
Set("ScriptPath", tmppath)
```

The `tmppath` variable will be set first, and is used to reset the ScriptPath as a final operation.

18.12 Static and Global Variables

Variables defined in script functions are automatic by default. The term "automatic" means that every call of the function provides a fresh set of variables. A static variable, on the other hand, retains its contents between calls, and the same variable storage is used in all calls to the function. One can explicitly assign a variable in a function to be static using the `static` keyword. This construct should appear only in functions (not the main procedure), and *must* appear ahead of all other executable statements. The syntax is

`static` *var1* `[=` *val*`]` *var2 ...*

The terms can be separated by white space and/or commas. The *var1*, etc., are variables used in the function that are to have static storage. They can optionally be initialized by including an assignment. If an assignment is used, the right hand side should consist of constants and variables that have already been assigned, meaning that they appear to the left in the present line or in a previous `static` line (there can be more than one). Array variables should have an initial dimensionality/size specification consisting of comma-separated integers enclosed in square brackets. Each such integer represents the maximum index for the dimension, with the lowest dimension listed to the left. This is the standard syntax for array declaration.

Example:

```
function myfunc(a, b, c)
static callcnt = 0
...
callcnt = callcnt + 1
Print("myfunc has been called", callcnt, "times")
endfunc
```

There is also provision for global variables. Global variables are variables whose scope extends to all functions where the variables have been declared, including the main procedure. These are useful for data items that are accessed frequently throughout a script application.

The `global` keyword is used to declare global variables. The syntax is identical to that for the `static` keyword, and similarly the declaration must appear at the top of a function and the main procedure. There can be more than one `global` line.

`global` *var1* [= *value*] *var2 ...*

In functions, the list following the keyword can not contain assignments or array subscripting. As with `static` declarations, `global` declarations must appear at the top of the function body. There can be multiple `global` lines, and these can be freely mixed with `static` lines. Global variables are not accessible unless declared.

A global variable must be declared in each function where it is to be accessed, and in the main procedure. Assignments and array initialization can be applied in the declarations in the main procedure only. It is an error to declare a global with assignment more than once, or to declare with an assignment in a function. Like other variables, if a global variable is not initialized in a declaration, the first assignment will define the variable type. Global array variables must be initialized with the maximum initial indices in each dimension, comma separated, enclosed in square brackets in the main function, but indices should *not* appear in the declarations in functions.

Example:

```
function myfunc()
    global gvar
    Print(gvar)
    gvar = gvar + 1
endfunc

global gvar = 1
myfunc()
Print(gvar)
# output is:
# 1
# 2
```

Global variables declared in functions create links to the global variable of the same name declared in the main procedure. If the function is defined in a separate file from the main procedure, such as a `library` file, and a global variable is declared and used in the function that is not also declared in the main procedure, an error results.

18.13 Predefined Constants

The following constants are recognized by name:

`e`	Natural log base
`pi`	3.14159...
`PI`	3.14159...
`NULL`	0
`INFINITY`	Maximum extent of object coordinate field
`TRUE`	1
`FALSE`	0
`EOF`	-1
`CHARGE`	1.60217646e-19
`CTOK`	273.15
`BOLTZ`	1.3806226e-23
`ROOT2`	square root of 2

The value of `INFINITY` is the internally assumed absolute limit for valid coordinates, in microns. This is 1e9 divided by the database resolution, which is the value of the DatabaseResolution variable if set, or the default of 1000.

18.14 HTML Forms and Scripts

HTML forms can be used as input devices for scripts. A form may provide a more convenient interface than a sequence of `Ask`*XXX* functions, and arbitrary text and links into the help system can also be provided in the form text.

18.14.1 Introduction to HTML Forms

Those interested in learning about forms in HTML should obtain a book on the subject. A decent book on writing HTML documents is

> HTML for the World Wide Web, Elizabeth Castro, Peachpit Press, Berkeley CA 1989, isbn 0-201-69696-7.

Below is a quick summary of the form-related tags.

An HTML form is a collection of input objects such as toggle buttons, text areas, and menus which allow the user to provide input. Within the form is a **submit** button, which when pressed causes a predefined action to occur. In HTML, the form is usually processed by the web page server (through a cgi script). In *Xic*, the form may instead be processed by an *Xic* script.

A form starts with a tag in the format

`<form method="post" action="`*some text*`">`

All but "*some text*" should be copied verbatim. In *Xic*, the "*some text*" is of the form

"`action_local_xic` *script_path*".

The quotes are required, and `action_local_xic` should be copied verbatim. The second word is the name of an *Xic* script file. The "`.scr`" extension is optional, and if a directory path is not given, the script should exist in the script search path. Often, *script_path* is the predefined macro `THIS_SCRIPT`, which is replaced by the name of or path to the present script.

The opening `<form ...>` tag is followed by the contents of the form itself, which can consist of formatted text, and references to the following objects.

Every object is given a unique name. This name is used to access the data in the script. Each button object will also have a value, which is a string token passed to identify a choice, i.e., which button of a group is selected. This may be different than the label on the button. In the tags, constructs like `name="`*name*`"` indicate the keyword `name`, followed by an '`=`' with no surrounding space, which is followed by a quoted text string.

Text Boxes

These are one-line entry areas. The tag format is

`<input type="text" name="`*name*`"` *options*`>`

The *options* (which are not required), can be:

`size="`*n*`"`
: The *n* is an integer that sets the field width in characters.

`maxlength="`*n*`"`
: The *n* is an integer which limits the length of input text.

`value="`*some text*`"`
: This indicates the text that will initially appear.

Example:

```
Enter username:  <input type="text" name="usertext">
```

In this and other similar elements that take a "`value="`*string*`"`" clause, note that this will fail if *string* contains quote ('`"`') characters. However, HTML '`&`' escapes are expanded in the string, so quote characters can be replaced with "`"`" to include quotation in the string.

Password Boxes

These are text boxes, except that characters are printed as '*' for security. The format is similar to text boxes:

```
<input type="password" name="name" options>
```

The *options* are the same as for text boxes.

Example:

```
Enter password:  <input type="password" name="passwd">
```

Radio Buttons

These are groups of buttons, one and only one of which is always selected. The tag format is

```
<input type="radio" name="name" value="value1" option>text
<input type="radio" name="name" value="value2" option>text
...
```

Each radio button in the group has a line of the form above. The only *option* is "`checked`" which can appear in only one line, and indicates which button is initially pressed (default is the first button listed). Each button should have the same `name`, and a different `value`. The text that follows the tag appears next to the button, and is usually but not necessarily the same as the value.

Example:

```
<input type="radio" name="radioset" value="1">1
<input type="radio" name="radioset" value="2" checked>2
<input type="radio" name="radioset" value="3">3
```

Check Boxes

These are toggle buttons. The tag format is

```
<input type="checkbox" name="name" value="value" option>text
```

The only option is "`checked`" which indicates that the button is initially pressed. The text following the tag appears next to the button, and is usually but not necessarily the same as the value.

Example:

```
<input type="checkbox" name="check1" value="check1">check1
<input type="checkbox" name="check2" value="check2" checked>check2
```

Text Blocks

These are multi-line text input areas. The tag format is

```
<textarea name="name" options>default text</textarea>
```

The options are

`rows="`*n*`"`
The *n* is an integer that sets the height to n characters.

`cols="`*n*`"`
The *n* is an integer that sets the width to n characters.

The *default text*, if any, will appear in the text area initially.

Example:

```
Type in your message:<br>
<textarea name="message" rows="12" cols="40">Dear sirs,
</textarea>
```

Option Menu

This is a menu of selections, shown as a button containing the current selection. Pressing the button produces a drop-down menu of choices. the tag format is

```
<select name="name" size="1">
<option value="value1" option>text
<option value="value2" option>text
...
</select>
```

There is one `<option ...>` tag per menu entry. The text following the `<option ...>` tag will appear in the menu. The only option is "`selected`" which can be given on only one line and indicates which item is initially selected (the default is the first item listed).

Example:

```
<select name="opmenu" size="1">
<option value="choose1">choose1
<option value="choose2">choose2
<option value="choose3">choose3
<option value="choose4" selected>choose4
</select>
```

Selection Menu

This type of menu has multiple lines, which can be selected by clicking. The menu may be scrollable. The tag format is

```
<select name="name" size="n" option>
<option value="value1" option>text
<option value="value2" option>text
...
</select>
```

The size in the `<select ...>` tag is an integer greater than 1, which indicates the number of lines visible. If this is less than the number of `<option ...>` lines that follow, the menu will be scrollable. The option that can appear in the `<select ...>` tag is "`multiple`" which if given allows multiple lines to be selected, otherwise only a single entry can be selected.

Example:

```
<select name="menu" size="2">
<option value="choose1">choose1
<option value="choose2">choose2
```

```
<option value="choose3">choose3
<option value="choose4">choose4
</select>
```

File Selection

This is a text area with a **browse** button. When the **browse** button is pressed, the **File Selection** panel appears, and the **Ok** button of the **File Selection** panel will transfer the selected file name to the form text area. The format of the tag is

```
<input type="file" name="name" option>
```

The only option is `size="n"` to set the width in characters of the text area.

Example:

```
<input type="file" name="filesel" size="64">
```

Each form must have a **submit** button. A **reset** button, which resets all objects to their initial state, is generally useful.

Submit Button

This is a button which initiates action on the form. This button is required if any action is to be taken on the form data. The tag format is

```
<input type="submit" option>
```

The only option is `value="message"`, where the *message* is the text that actually appears on the button, which is "Submit" if no value is specified.

Example:

```
<input type="submit" value="Done">
```

Reset Button

This button resets each component of the form to the initial state. The tag format is

```
<input type="reset" option>
```

The only option is `value="message"`, where the *message* is the text that actually appears in the button, and is "Reset" if no value is specified.

Example:

```
<input type="reset">
```

The form items can be intermixed with text, images, or other HTML formatting and objects. To terminate a form definition, one must supply the tag

```
</form>
```

Below is the "`spform.html`" file which is used with the **spiralform** demonstration script, as an example.

```
<h2>Forms Demo -- Generate a Spiral</h2>
This page demonstrates the use of HTML forms as input devices for
<i>Xic</i> <a href="xicscript">scripts</a>.  Press the <b>Submit</b>
button when ready.  The spiral will be attached to the pointer, and
can be placed by clicking in a drawing window.

<p>
<form method="post" action="action_local_xic spiralform">

Choose the number of turns in the spiral
<select name="opmenu" size="1">
<option value="1">1
<option value="2">2
<option value="3">3
<option value="4">4
<option value="5">5
<option value="6">6
<option value="7">7
<option value="8">8
<option value="9">9
</select>

<p>
Enter the path width: <input type="text" name="pwidth" value="4"><br>
Enter the starting radius: <input type="text" name="rad1" value="20"><br>
Enter the pitch: <input type="text" name="pitch" value="10"><br>

<p>
Select the number of edges per turn:
<input type="radio" name="radioset" value="10" checked>10
<input type="radio" name="radioset" value="20">20
<input type="radio" name="radioset" value="40">40

<p>
<input type="submit">
<input type="reset">
</form>
```

18.14.2 Interfacing Forms to *Xic* Scripts

If a form has an action which is in the format "`action_local_xic` *scriptname*" then, when the **submit** button is pressed, the script in *scriptname* will be called, with the following:

- The preprocessor variable SUBMIT is defined.
- A string-type variable is created for each active form element. The variable name is that of the `name` field in the form element tag (so these must be unique). The value of the variable is from the `value` tag of the selected button, or the text of a text-entry object. The variable is defined only if the text object had text, or if a check button was pressed.

Within the script, one must supply the following logic:

- Determine if the SUBMIT preprocessor variable is defined. If yes, than the script was called by a form, otherwise the script was called by a button in the **User Menu**. Note that this enables the script to initiate showing the form, as will be seen in the example below.
- For each variable, the script must identify if the variable was set, i.e., a text entry had text, and possibly convert the text to a numeric value. The input should also be sanity checked at this point.

Below are the first few lines of an example script which could interface to the example form given above. When the script is selected in the **User Menu**, it will display the help window containing its input form. When the **submit** button of the form is pressed, the script will be called again, and the data processed.

```
#ifndef SUBMIT
# SUBMIT is not defined, so we are being called from the User Menu
# pop-up our input form in the HTML viewer and exit
TextCmd("help spform.html")
Exit()
#endif

# We are being called from the form (SUBMIT is defined)
# First check the option menu return.  The entries are digits, which must
# be converted from text strings to real values
#
if Defined(opmenu)
    num = ToReal(opmenu)
else
# This should never fail, since the option menu always has a selection
    ShowPrompt("number of turns unknown")
    Exit()
end

# Next check the return from a text entry object.  Exit if the variable
# is undefined (text input empty), or the result is a bad numeric value
#
if Defined(pwidth)
    width = ToReal(pwidth)
    if (width < 0)
        ShowPrompt("Bad input (< 0) for width: ", width)
        Exit()
    end

    ShowPrompt("width unknown")
    Exit()
end

(check other input variables)
(perform calculations/operations)
Exit
```

This is typical boilerplate for a form-entry script.

18.15 Example Script

Below is the text of a script which will generate a spiral object, provided as an example.

```
# Example script to produce a spiral on the current layer
#
# solicit geometrical info from user
num = AskReal("Number of turns? ", "1")
if (num < 0 | num > 100)
    ShowPrompt("Bad input (< 0 or > 100) for turns: ", num)
    Exit()
end
width = AskReal("Width of spiral path? ", "4")
if (width < 0)
    ShowPrompt("Bad input (< 0) for width: ", width)
    Exit()
end
rmin = AskReal("Starting radius? ", "20")
if (rmin < width/2)
    ShowPrompt("Bad input (< width/2) for min radius: ", rmin)
    Exit()
end
spa = AskReal("Pitch? ", "10")
if (spa < width)
    ShowPrompt("Bad input (< width) for pitch: ", pitch)
    Exit()
end
nums = AskReal("Edges per 360 degrees? ", "50")
if (nums < 3 | nums > 90)
    ShowPrompt("Bad input (< 3 or > 90) for edge count: ", nums)
    Exit()
end

# initialize
width = width/2
dth = 2*pi/nums
n = nums*num + 1
i = 0
theta = 0

# there is an internal limit of 2000 polygon vertices
nverts = 2*n + 1
if (nverts > 2000)
    ShowPrompt("Sorry, resulting polygon would have too many vertices.")
    Exit()
end

# allocate array, size 2*nverts
array[4000] = 0

l = 4*n
```

```
j = 0

# fill in the array
while (i < n)
    r = rmin + theta*spa/(2*pi)
    x = (r-width)*cos(theta)
    y = (r-width)*sin(theta)
    array[j] = x
    array[j+1] = y
    x = (r+width)*cos(theta)
    y = (r+width)*sin(theta)
    array[l-j-2] = x
    array[l-j-1] = y
    j = j + 2
    i = i + 1
    theta = theta + dth
end

# close the path, necessary for polygon
array[l] = array[0]
array[l+1] = array[1]

# get the location for the spiral and transform array
ShowPrompt("Point to locate center of spiral")
xy[2]
PushGhost(array, nverts)
ShowGhost(8)
if !Point(xy)
    Exit()
end
ShowGhost(0)
PopGhost()

i = 0
j = 0
while (i < nverts)
    array[j] = array[j] + xy[0]
    array[j+1] = array[j+1] + xy[1]
    i = i + 1
    j = j + 2
end

# create the polygon
drc = DRCstate(0)
Polygon(nverts, array)
Commit()
DRCstate(drc)
ShowPrompt("Info: spiral not drc'ed.  Drc takes a long time for these objects.")

#done
```

Chapter 19

Keyboard '!' Commands

The command line interface through the prompt area provides an interface to operating system commands, as well as to a number of internal commands which are often rather specialized and are not associated with a menu button. Each of these commands starts with an exclamation point "**!**", and may be entered when no other command is active, or inside of many commands. These key presses are not recorded in the "keys" area below the side menu. If the command entered matches one of the internal commands listed below, that command is executed. Otherwise, an operating system shell and associated window is produced to execute the command, with the exclamation mark stripped.

Special Form: !
Entering a single exclamation point with no other text will produce an interactive terminal window into which the user can issue operating system commands. If any text follows the exclamation point, and that text does not match an internal command, the exclamation point will be stripped, the remaining text sent to the operating system for execution, and the result will be displayed in a pop-up window.

Giving the bare exclamation point is equivalent to giving the **!shell** command without arguments (see 19.23.1). Giving something like `!xyz` is equivalent to giving `!shell xyz`, provided that `!xyz` is not one of the built-in commands. The use of `!shell` removes the ambiguity.

Special Form: !!
If a line starts with "**!!**", the rest of the line is taken as a script, and executed by the script parser. This is how to map script interface functions into a macro. For example, below is a macro to reset the current transform:

`!!SetTransform(0,0,1)` **Ctrl-Return**

Special Form: !?
Entering "**!?**" will bring up help about the '**!**' commands.

Special Form: !?*name*
This special form will bring up help about the help database keyword *name*.

Special Form: !??
This special form will print a listing of the '!' commands actually available in the program, from internal tables.

Special Form: !#
The last six commands given are saved, and can be recalled with the form "**!#**[n]", that is, an exclamation point and a pound sign followed by an optional integer. The n is an optional integer 0–5, and if not given

(the square brackets indicate "optional" and are not literal) a value of 0 is taken. The n'th previous command will be printed in the prompt area, where it can be edited and re-executed. If no matching command is found, there is no action.

When a command from the history list is in the prompt area, the **Up Arrow** and **Down Arrow** keys can be used to cycle through the other commands in the history list, each of which will be entered into the prompt line in response to the key press.

Each '**!**' command given, including those from '**!#**', will be pushed onto the history list in the 0 position if it is not identical to the previous command given.

The following table summarizes the internal commands available. These commands are described in detail in the following sections.

Compression	
!gzip	Apply compression to file
!gunzip	Uncompress a file
!md5	Print file digest
Create Output	
!sa	Save modified cells
!sqdump	Save selections as native cell
!assemble	Process or merge archive files
!splwrite	Split a layout into multiple pieces
Current Directory	
!cd	Change working directory
!pwd	Print working directory
Diagnostics	
!time	Print elapsed run time in seconds in console
!timedbg	Print timing info in console
!xdepth	Print transform stack depth in console
!bincnt	Database diagnostic
!netxp	Net expression checking
!pcdump	Parameterized cell database dump
Design Rule Checking	
!showz	Show DRC partitioning
!errs	Rebuild DRC error highlighting from file
!errlayer	Create error polygons on some layer
Electrical	
!calc	Calculate parameter expression value
!check	Check electrical input for consistency
!regen	Regenerate damaged file
!devkeys	Print device key table
Extraction	
!antenna	Test for MOS antenna effect
!netext	Batch physical net extraction
!addcells	Add missing cells
!find	Find devices
!ptrms	Physical terminal manipulations
!ushow	Show unassociated elements
!fc	Control capacitance extraction interface
!fh	Control inductance/resistance extraction interface
Graphics	
!setcolor	Set attribute color
!display	Display graphics in a foreign X window
Grid	
!sg	Save the current grid
!rg	Restore saved grid
Help	
!help	Call the help system
!helpfont	Set help base font family

!helpfixed	Set help fixed font family
!helpreset	Clear help topic cache
Keyboard	
!kmap	Read keyboard mapping file
Layers	
!ltab	Manipulate layer table
!ltsort	Alphanumerically sort layer table
!exlayers	List layers by applied keywords in console
Layout Editing	
!array	Manipulate instance arrays
!layer	Create layers/objects using expression
!mo	Move objects
!co	Copy objects
!spin	Rotate objects
!rename	Rename subcells
!svq	Save selections in register
!rcq	Recall selections from register
!box2poly	Convert boxes to polygons
!path2poly	Convert wire paths to polygons
!poly2path	Convert polygon boundaries to wires
!bloat	Expand/contract object
!join	Join objects into polygon
!jw	Join similar wires with common endpoints
!split	Split polygon into trapezoids
!manh	Convert to Manhattan polygons
!polyfix	Fix polygon errors
!polyrev	Reverse polygon winding
!noacute	Eliminate acute angles
!togrid	Move selected object vertices to grid
!tospot	Condition object for spot size
!origin	Set origin of current cell
!import	Import structures into the current cell
Layout Information	
!fileinfo	Print info about archive file in console
!summary	Print summary info of current hierarchy
!compare	Compare geometry in files
!diffcells	Create cells from **!compare** output
!empties	Check for empty cells
!area	Measure object area
!perim	Measure object perimeter
!bb	Print bounding box of current cell
!checkgrid	Check object for off-grid vertices
!checkover	Report cells that overlap
!check45	Select polygons and/or wires with angle non-45 degree multiple
!dups	Select coincident identical objects
!wirecheck	Check wire characteristics
!polycheck	Check polygon characteristics
!polymanh	Select Manhattan polygons

!poly45	Select polygons with angle not a 45 degree multiple
!polynum	Show polygon vertex indices
!setflag	Set cell flags
Libraries and Databases	
!mklib	Create or append to a library file
!lsdb	List "special" databases in memory
Marks	
!mark	Create user marks in layout
Memory Management	
!clearall	Clear all memory
!vmem	Windows only, print virtual memory statistics
!mmstats	print memory manager statistics
!mmclear	Clear caches
OpenAccess Interface	
!oaversion	Print OpenAccess release number
!oadebug	Enable log files
!oanewlib	Create new OpenAccess Library
!oabrand	Permit save from *Xic* in OA library
!oatech	Query OpenAccess technology database
!oasave	Save cell to OpenAccess library
!oaload	Read cell from OpenAccess library
!oadelete	Delete OpenAccess object
Parameterized Cells	
!rmpcprops	Remove pcell properties from pcell sub-masters
!preload	Pre-load sub-masters to resolve as cell data are read
Rulers	
!dr	Delete rulers
Scripts	
!script	Add a script to the **User Menu**
!rehash	Re-read script libraries and rebuild **User Menu**
!exec	Execute a script
!lisp	Execute a Lisp script
!py	Execute a Python script
!tcl	Execute a Tcl script (no Tk)
!tk	Execute a Tcl/Tk script
!listfuncs	Pop-up list of saved functions
!rmfunc	Delete a saved function
!mkscript	Create script that generates current cell hierarchy
!ldshared	Load a script interface plug-in
Selections	
!select	Select objects
!desel	Deselect objects
!zs	Zoom to selected objects
Shell	
!shell	Open terminal window
!ssh	Open terminal window to remote system
Technology File	
!attrvars	List the variables that are recognized as tech file keywords

!dumpcds	Create Cadence Virtuoso^{TM} technology and DRF files
Update Release	
!update	Download/install new release
Variables	
!set	Set/examine variables
!unset	Unset variables
!setdump	Dump variables
***WRspice* Interface**	
!spcmd	Execute *WRspice* command

19.1 Compression

19.1.1 The !gzip Command: Compress Files

Syntax: `!gzip` *infile* [*outfile*]

The will compress the file given as *infile* using the `gzip` method. If *outfile* is not given, output is written to a file with the same name as *infile* but with a "`.gz`" extension. Otherwise, the file name given for *outfile* must have a "`.gz`" extension. Under Unix/Linux this uses 64-bit file offsets so can be applied to files larger than 2Gb, unlike some versions of the GNU `gzip` utility. Unlike the GNU `gzip` program, this will not delete *infile*.

19.1.2 The !gunzip Command: Uncompress Files

Syntax: `!gunzip` *infile* [*outfile*]

This will uncompress the file given as *infile*, which was previously compressed with `gzip`, and has a "`.gz`" extension. If no *outfile* is given, output is written to a file with the same name as the *infile* but with the "`.gz`" suffix stripped. Under Unix/Linux this uses 64-bit file offsets so can be applied to files larger than 2Gb, unlike some versions of the GNU `gunzip` utility. Unlike the GNU `gunzip` program, this will not delete *infile*.

19.1.3 The !md5 Command: Print File Digest

Syntax: `!md5` *filepath*

This command will compute and print on the prompt line the MD5 digest of a file. The digest is a sequence of 32 hex digits that is very unlikely to duplicate that of another file. It can be used to determine if a file is incorrect or has been tampered with.

The same digest can be obtained from the following command, which is available on most Unix/Linux/macOS systems:

`openssl dgst -md5` *filepath*

19.2 Create Output

19.2.1 The !sa Command: Save Modified Cells

Syntax: `!sa`

Invoking this command is the same as invoking the **Save** command in the **File Menu**. If there are modified cells, the **Modified Cells** pop-up will appear, from which the cells can be saved to disk.

19.2.2 The !sqdump Command: Save Selections as Native Cell

Syntax: `!sqdump` *cellpath*

This will save the current selections to a native file named in *cellpath*. Unlike the **Create Cell** command in the **Edit Menu**, no cell is created in memory.

19.2.3 The !assemble Command: Merge Archives

Syntax: `!assemble` *specfile* — *argument_list*

The **!assemble** command automates reading of cells from archives, subsequent processing, and writing to a new archive file. It provides the capabilities of the **Format Conversion** panel in the **Convert Menu**, such as format translation, windowing, and flattening. Additionally, multiple input files and cells can be processed and merged into a larger archive, on-the-fly or by using a Cell Hierarchy Digest (CHD) so as to avoid memory limitations. Cell definitions for the read and possibly modified cells are streamed into the output file, and the output file can contain a new top-level cell in which the cells read are instantiated. The input and output can be any of the supported archive formats, in any combination.

The operation can be controlled by a specification script file, the path to which is given as the argument. The script uses a language that is unique to this command, which will be described. This supplies the output file name and the description of the top-level cell (if any), the files to be used as input, the cells to extract from these files, and the operations to perform. It is a simple text file, prepared by the user, containing a number of keywords with values. The specification script can also be obtained from the **Assemble** command in the **Convert Menu**, which is a graphical front-end to the **!assemble** command.

Alternatively, the argument list can consist of a series of option tokens and values. These are logically almost equivalent to the language of the specification file. This gives the user the option to enter job descriptions entirely from the command line. These command-line options start with a '`-`' character. If the first argument given starts with '`-`', a list of option arguments is assumed, otherwise the argument is taken as a file name. If the specification file name starts with '`-`', one should prepend the name with "`./`" to avoid a parse error.

Only physical data are read, electrical data will be stripped in output. A log file is produced when the **!assemble** command is run. If not explicitly set with a `LogFile`/`-log` specification, this is named "`assemble.log`" and is written in the current directory. The log file contains warning and error messages emitted by the readers during file processing, and should be consulted if a problem occurs.

File and Option Argument Format

The **!assemble** command parses and executes a specification file or option list in the format described below. The file text contains keyword directives and values which specify the operations to be performed. Each active line begins with a keyword, and all keywords are case-insensitive. Blank lines and lines that begin with non-alpha characters are taken as comments and are ignored. Unrecognized tokens will generate an error and no processing will be done. There is an almost one-to-one correspondence between file keywords and equivalent command-line options. For options that require a string, the string can be double-quoted (`"..."`), and these **must** be quoted if they contain white-space.

The command input can either come from a file, or from the command-line arguments, but not both.

Overall, the input logically contains three levels of directives:

```
Header Block
Source Block
        [Placement Block]
        [...]
[...]
```

The Header Block contains a mandatory output file specification line, and optional additional lines. The Source Block contains a reference to a source file, and may contain zero or more Placement Blocks, which identify a particular cell from that file. The specification must contain at least one Source Block.

Indentation can be used in the specification file to highlight the scoping. The same logic applies in an argument list, but may be less visible since all options appear in one line.

Header Block

The Header Blocks contains global directives. This must be followed by at least one Source Block, which specifies an input source.

`OutFile` *out_file_name*
(option: `-o` *out_file_name*)
This line or option is mandatory, and provides the name of the file to be used for output. This must appear before any Source Blocks. The output file name **must** have a recognized extension that corresponds to the format to be used. These are:

CGX	`.cgx`
CIF	`.cif`
GDSII	`.gds, .str, .strm, .stream`
OASIS	`.oas`

Only these extensions are recognized, however CGX and GDSII allow an additional `.gz` which will imply compression.

Basic defaults for the various output formats are as specified in the **Export Control** panel from the **Convert Menu**, or from the corresponding variables.

`LogFile` *logfile*
(option: `-log` *logfile*)
This specifies the name of a log file which is produced during the run. This will record messages, warnings, and errors that are emitted. If not given, a log will be written using a default file name, which is "`assemble.log`" in the current directory, for the **!assemble** command.

`TopCell` *cellname*
(option: `-t` *cellname*)
This optional line or option specifies that a new top-level cell is to be created in output. At most one `TopCell` can be given. This must appear before any Source Block.

If a `TopCell` is given, a corresponding cell definition will be created in the output file, and all cells specified in Placement Blocks (the "placements") will be instantiated in the new cell. Whether or not a `TopCell` is given, the placements will be streamed to the output file, meaning that the cell definitions needed to describe the cell and possibly its hierarchy will be added to the output file. With a `TopCell` given, the placements will be instantiated in the new top cell in output. Otherwise, there is no placement, and redundant Placement Blocks will be ignored. The output file can end up with multiple top-level cells, which may be desirable when creating a library.

The Header can also contain any of the Source Block or Placement Block directives below. These will be used as defaults in all blocks that follow, but can be overridden from within the blocks, or set, modified, or reset between Source Blocks.

Source Blocks

The Source Blocks specify an input file or CHD, and provide directives that are active when the source is read. The Source Block may contain Placement Blocks, which identify individual cells or cell hierarchies to be read.

The same file might be used in more than one Source Block, if the directives, such as cell name modification, are different in the two blocks.

The Source Blocks start with the following keyword:

`Source` *filename*
(option: `-i` *filename*)
This line or option represents the start of a Source Block for the given input file. The file must be in one of the supported archive formats, and the format is recognized automatically, so there is no name suffix requirement as with the output file name.

The absence of any Placement Blocks defined in the Source Block implies that all cells found in the file will be read.

The *filename* can also be the access name of a CHD which already exists in memory. In this case, the CHD is used for access, and cell names given in Placement Blocks must include any cell name mapping which is used in the CHD.

Further, the *filename* can be that of a CHD saved to disk, such a with the **Save** button in the **Cell Hierarchy Digests** panel. In this case, the CHD will be read into memory, and used as the source.

In any case where a Source Block contains a Placement Block, a temporary CHD will be created anyway if one is not given, so explicitly naming the CHD may save time/space in some cases.

In cases where a CHD is named, but no Placement Blocks are given, the hierarchy of the CHD's default cell will be streamed. The default cell is the first top-level cell found in the file, or can be configured into the CHD.

The Source Blocks can be terminated with:

`EndSource`
(option: `-i-`)
This optional keyword or option terminates the present Source Block. Lines or text tokens that follow, up to another `Source` keyword or `-i` option, are taken in the context of the Header Block. Thus, directives can be set, modified, or reset between Source Blocks, and will remain in force (in the Header Block context) until reset or modified between subsequent Source Blocks. This keyword is optional, as it is implicit if another `Source` line or `-i` option is given. It is required only if one wishes to change the directives in the Header context for subsequent Source Blocks.

Within the Source Block, one may find Placement Blocks, Source Block directives, and Placement Block directives.

Source Block Directives

The Source Block directives can be given in the context of the Header Block, in which case they serve as defaults for the Source Blocks that follow. They can also be given in a Source Block, in which case they apply in that Source Block only, and override a similar directive active from a definition in the Header Block context. The term "Header Block context" means that the definition appears before any Source Block, or after an `EndSource` line (`-i-` option) but before the next `Source` line (`-i` option.

The Source Block directives can not appear inside of Placement Blocks, where they would have no meaning. Thus, in a Source Block, Source Block directives can appear before the Placement Blocks, or between `EndPlace` lines (`-c-` option) and the next `Place` (`-c` option) or `PlaceTop` line (`-ctop` option). The directives that apply are those logically in force at the end of the Source Block. The Source Block directives apply to the Source Block, and will have the same effect for all contained Placement Blocks, regardless of ordering.

The following lines define Source Block directives:

`LayerList` *list_of_layer_names*
(option: `-l` *list_of_layer_names*)
This saves a list of space-separated layer names or hex-encoded pseudo-names to be used with the layer filtering directives `OnlyLayers` (`-n` option) and `SkipLayers` (`-k` option). This directive in itself does not alter output. This list is implied when a *list_of_layer_names* is provided with these keywords. In the command line, the list of layer names must be quoted if it contains more than one entry, but this is not required in a file.

`OnlyLayers` [*list_of_layer_names*]
(option: `-n`)
When active, only the listed layers will be used in output, geometry on other layers will be skipped. Arguments following this keyword will be used to set or reset the `LayerList`, and have the same interpretation as for that keyword. If no arguments follow, the `LayerList` currently in scope will be used. The `-n` command line token *does not* accept a list of layer names, unlike the corresponding keyword. This must be separately specified with a `-l` option.

`NoOnlyLayers`
(option: `-n-`)
Turn off restriction to layers in the `LayerList`, if the `OnlyLayers` directive (`-n` option) is in force. The corresponding `LayerList` remains defined.

`SkipLayers` [*list_of_layer_names*]
(option: `-k`)

When active, listed layers will not appear in output, geometry on layers not listed will appear in output. Arguments following this keyword will be used to set or reset the `LayerList`, and have the same interpretation as for that keyword. If no arguments follow, the `LayerList` currently in scope will be used. The `-k` command line token *does not* accept a list of layer names, unlike the corresponding keyword. This must be separately specified with a `-l` option.

`NoSkipLayers`
(option: `-k-`)
Turn off layer skipping, if the `SkipLayers` directive (`-k` option) is currently in force. The associated `LayerList` remains defined.

`LayerAliases` *name1=alias1 name2=alias2 ...*
(option: `-a` *name1=alias1 name2=alias2 ...*)
This keyword provides a list a layer aliasing definitions to apply in output. The layer names can be hex-encoded pseudo-names when applicable. This is similar to the layer aliasing found in the **Format Conversion** panel and elsewhere. In the command line, the list must be quoted if it contains more than one entry, but this is not required in a file.

`ConvertScale` *scale_factor*
(option: `-cs` *scale_factor*)
This directive has effect only in the case where there are no Placement Blocks, and is ignored otherwise. This will scale all coordinates read from the source by the given factor, which can be in the range 0.001 through 1000.0. Thus, in output, the corresponding cell definitions will be scaled by this factor. This is similar to the `Scale` Placement Block directive (`-s` option), but applies when there are no Placement Blocks and Placement Block directive are ignored.

`ToLower`
(option: `-tlo`)
This sets a flag to indicate conversion of upper case cell names to lower case in output. Mixed-case cell names are unaffected.

`NoToLower`
(option: `-tlo-`)
Turn off lower-casing, if the `ToLower` directive (`-tlo` option) is currently in force.

`ToUpper`
(option: `-tup`)
This sets a flag to indicate conversion of lower case cell names to upper case. Mixed-case cell names are unaffected.

`NoToUpper`
(option: `-tup-`)
Turn off upper-casing, if the `ToUpper` directive (`-tup` option) is currently in force.

`CellNamePrefix` *prefix_string*
(option: `-p` *prefix_string*)
Cell name change prefix. This operation occurs after case conversion. The *prefix_string* is interpreted in the manner of the InCellNamePrefix variable.

`CellNameSuffix` *suffix_string*
(option: `-u` *suffix_string*)
Cell name change suffix. This operation occurs after case conversion. The *suffix_string* is interpreted in the manner of the InCellNameSuffix variable.

Placement Blocks

Placement Blocks can appear only within Source Blocks. Each Source Block can have zero or more Placement Blocks. If no Placement Blocks are given, all cells in the source file are written to output, and Placement Block directives that may be in force are ignored. If the Source Block specifies a CHD source, absent any Placement Blocks, the hierarchy of the CHD's default cell will be streamed to output.

A Placement Block is used to indicate a specific cell within the source file, which will be written to output. The Placement Block directives specify actions to take, for example whether to process just this cell or its hierarchy, whether to use flattening and/or windowing, and the placement transform if the cell is to be instantiated in a given `TopCell`.

As cells are written to output, a table is maintained to prevent writing duplicate cell definitions. Each cell needed to represent the cell hierarchies contained in the output file is written once only. When different versions of the same cell are needed, such as with different scaling, the names of the cells are altered to avoid a name clash. This is accomplished by appending "`$`*N*", where *N* is an integer which makes the new name unique, to the cell names.

A new Placement Block, which can appear only within a Source Block, will begin with either of the following keywords or options:

`Place` *cellname* [*placement_name*]
(option: `-c` *cellname*)
The *cellname*, which must name a cell in the source file, will be included in the output file. If a `TopCell` was given, the cell will also be instantiated in the given top cell. The *placement_name*, if given, will replace *cellname* in output. In either case, any cell name alteration presently in force will be applied. If a Placement Block matches a previous block except for the transformation parameters (`Translate`, `Rotate`, `Magnify`, `Reflect`), then if a `TopCell` was given, an instance will be added with the new transform, but the cell definitions are already in the output and will not be streamed. Thus, in this case with no `TopCell`, there would be no addition to output.

In a command line, the *placement_name* can not follow the *cellname* as in a file. Rather, there is a special option token

`-ca` *placement_name*

that can appear within the Placement Block which specifies the name change.

`PlaceTop` [*placement_name*]
(option: `-ctop`)
The `PlaceTop` line (`-ctop` option) is equivalent to a `Place` line (`-c` option), except that it will automatically select the first top-level cell found in the source. It is equivalent to the `Place` line (`-c` option) with the name of this cell as the first (only) argument. This is convenient when the top-level cell name is unknown. Unlike the keyword, the `-ctop` option does not take a following *placement_name*, which must be given by a `-ca` option within the Placement Block.

A Placement Block can be terminated with:

`PlaceEnd`
(option: `-c-`)
This optional keyword will end the current Placement Block. Subsequent lines will be accepted in the scope of the containing Source Block. This keyword is optional, as it is implicit if a `Place` or `PlaceTop` keyword (`-c` or `-ctop` option) is given. It is useful if one needs to add, modify, or reset

Placement Block directives in the Source Block scope, which will apply to subsequent Placement Blocks.

A Placement Block may contain any of the Placement Block directives, which control how the cell is treated in output. The transformations apply only when a `TopCell` was given in the Header Block, and control the location and orientation of the instantiation.

Placement Block Directives

The Placement Block directives can appear in the Header Block context, the Source Block context, or within a Placement Block. Thus, they can appear virtually anywhere in the specification file or command line, though the location alters the scope.

If given in the Header Block context, meaning that the directive appears before the first Source Block, or after an `EndSource` line (`-i-` option) but ahead of the next `Source` line (`-i` option), then the directive will be active as a default in all Source Blocks that follow, until the directive is changed or reset in the Header Block context.

Similarly, if a Placement Block directive is given in a Source Block, it will override a similar directive set in the Header Block scope, and will apply to all Placement Blocks that follow within the Source Block, until changed or reset in the context of the same Source Block. Being given in a Source Block, or in the context of a Source Block, means that the directive appears before the first `Place` or `PlaceTop` line (`-c` or `-ctop` option), or after an `EndPlace` line (`-c-` option but before the next `Place` or `PlaceTop` line or equivalent options.

If the Placement Block directive appears within a Placement Block, it will override a similar directive set in the Source Block or Header Block, and will apply to the current Placement Block only.

Placement Block directives are ignored when reading a source that has no Placement Blocks.

The following directives define the transformation applied to an instantiation of the cell in the `TopCell`. These will be ignored unless a `TopCell` was given.

`Translate` *x y*
: (options: `-x` *x* `-y` *y*)
Specify the translation coordinates. If not given, the default is 0, 0. Note that the keyword corresponds to two command-line options.

`Rotate` *angle*
: (option: `-ang` *angle*)
Specify a rotation angle, which must be a multiple of 45 degrees. If not given, the default is no rotation.

`Magnify` *magn*
: (option: `-m` *magn*)
Specify an instance magnification. If not given, the default is 1.0. Values from .001 to 1000.0 are accepted.

`Reflect`
: (option: `-my`)
Apply a mirror-Y transformation (before rotation, if any).

`NoReflect`
: (option: `-my-`)
Turn off the mirror-Y transformation, if the `Reflect` directive (`-my` option) is currently in force.

The following directives initiate operations on the cell definition, as it is written to output. These are performed whether or not a `TopCell` was defined.

`Scale` *scale_factor*
(option: `-s` *scale_factor*)
The cells read from the source will have all coordinates multiplied by the scale factor, which can be in the range .001 – 1000.0. This is distinct from the `Magnify` factor, which applies only to the instance created in the `TopCell`, and will in effect multiply the scale factor. When there are no Placement Blocks, and so Placement Block directives are ignored, the `ConvertScale` Source Block directive (`-cs` option) can be used to obtain the same effect.

`NoHier`
(option: `-h`)
If given, only the specified cell is written to output, and not its complete hierarchy as is the normal case. This can produce output files with unresolved subcell references, which must be satisfied by some means.

`NoNoHier`
(option: `-h-`)
Turn off the no-hierarchy mode, if the `NoHier` directive (`-h` option) is currently in force.

`NoEmpties [`*N*`]`
(option: `-e[`*N*`]`)
These enable various permutations of the empty cell filtering operations, as described for the **Format Conversion** panel in 14.10. These are:

"`NoEmpties`" or "`NoEmpties 1`"
(option: "`-e`" or "`-e1`")
Turn on both pre- and post-filtering.

"`NoEmpties 2`"
(option: "`-e2`")
Turn on pre-filtering only.

"`NoEmpties 3`"
(option: "`-e3`")
Turn on post-filtering only.

"`NoNoEmpties`" or "`NoEmpties 0`"
(option: "`-e-`" or "`-e0`")
Turn off all empty cell filtering.

`NoNoEmpties`
(option: `-e-`)
Turn off empty cell filtering, if the `NoEmpties` directive (`-e` option) is currently in force (above). These have synonyms "`NoEmpties 0`" and "`-e0`".

`Flatten`
(option: `-f`)
If given, all geometry under the cell being read will be written as part of the cell being read, i.e., the cell hierarchy will be flattened. The `NoHier` directive (`-h` option) is ignored if this is active.

`NoFlatten`
(option: `-f-`)
Turn off flattening, if the `Flatten` directive (`-f` option) is currently in force.

`Window` *left bottom right top*
(option: `-w` *left*,*bottom*,*right*,*top*)
If given, only the subcells (if `NoHier` is not active) and objects needed to describe the given area in the cell being placed will be written. The coordinates apply to *cellname* after any scaling is applied, and are given in microns. The four numbers can be separated by commas and/or white space. In the command line, if white space is present between numbers, the four numbers must be quoted, but this is not required in a file.

`Clip`
(option: `-cl`)
If `Window` was given, this will cause geometry to be clipped to the window.

`NoClip`
(option: `-cl-`)
Turn off clipping, if the `Clip` directive (`-cl` option) is currently in force.

19.2.4 The !splwrite Command: Split an Archive

Syntax: `!splwrite` `-i` *filename* `-o` *basename.ext* [`-c` *cellname*] `-g` *gridsize* `|` `-r` *l*,*b*,*r*,*t*[,*l*,*b*,*r*,*t*]...
[`-b` *bloatval*] [`-w` *l*,*b*,*r*,*t*] [`-f`] [`-m`] [`-cl`] [`-e`[*N*]] [`-p`]

This command will write output files corresponding to a list of rectangular regions, or to the partitions of a square grid logically covering all or part of a specified cell in a given layout file. The output files contain physical data only. These files can be flat or hierarchical.

The arguments are as follows:

`-i` *filename*
This mandatory argument specifies a path to a layout file, the access name of a Cell Hierarchy Digest (CHD) in memory, or a path to a saved CHD file. This source will provide cell data as input.

`-o` *basename.ext*
This mandatory argument provides the base name of the output files that will be created, and the type of file to write. There are generally two components of the argument, separated by a period. The *basename* component may be absent, but the period must remain. If the *basename* is absent, the name of the top-level cell being split will be used.

The *ext*, which follows the period, must be one of the following to indicate the file format to be used for output.

CGX	`.cgx`
CIF	`.cif`
GDSII	`.gds, .str, .strm, .stream`
OASIS	`.oas`

The GDSII and CGX extensions can be followed by "`.gz`", which will indicate `gzip` compression.

When writing a list of regions, the file names produced will have the form

basenameN.ext

where *N* is a 1–based index of the region in the order given. When writing grid cells, the file names produced will have the form *basename_X_Y.ext*

where X and Y are the 0–based indices of the corresponding grid cell (the origin is the lower-left corner).

`-c` *cellname*
: This optional argument specifies the name of the cell to be used as the top-level in output. If not given, this will be the first top-level cell found in the input file, or, if the input source is a CHD, the default cell configured into the CHD will be used.

Exactly one of the following two options must be provided.

`-g` *gridsize*
: This argument specifies the length, in microns, of the side of a square grid cell. The area to be written will be tiled with a grid of this size, with the origin at the lower left corner. Each grid cell with nonzero overlap area with the area to be written will have a corresponding output file produced.

`-r` *l*,*b*,*r*,*t*[,*l*,*b*,*r*,*t*]...
: This provides a list of rectangular regions to write, as a comma-separated list of coordinates in microns. Each region is specified by four coordinates in the order given, with no white space.

The regions can be given with a single `-r` followed by any number of concatenated regions, as implied above. However, any number of `-r` options with region lists can be given, the regions will be processed in order. Some users may find it more convenient to specify the regions individually, each with a separate `-r` option.

`-b` *bloatval*
: This optional argument specifies how much, in microns, the grid cells will be bloated before the write operation. If positive, the grid cells will be expanded, and the files will logically overlap. The value can also be negative, which will leave logically unwritten area between output files.

If a region list is specified rather than a grid, the bloating will be applied to each region.

`-w` *l*,*b*,*r*,*t*
: This specifies a rectangular area, in the top-level cell being written, which will be included in the output files. The four numbers are given in microns, separated by commas, with no intervening white space. If not provided, the entire cell area is understood.

`-f`
: If this flag is given, the output files will be flat. All geometry will be contained in the top-level cell of each file. Be aware that this can consume a lot of disk space.

If not given, the output files will maintain the hierarchy of the original file. In this mode, only the geometry needed to fully render the area of the top-level cell corresponding to the (possibly bloated) grid cell area is retained. Subcells may therefor contain only part of the original geometry, or may not appear at all if not instantiated within the area. Subcells may also become empty, these are not automatically stripped.

`-m`
: If flattening, this option specifies that a suffix "_N" is added to the top cell name in each file, with N an integer, so as to make the cell names unique in the collection. This will facilitate subsequent merging of data from the files by avoiding cell name clashes. Without this option, the files would have the same cell name, the same name as the original top-level cell. This option is ignored if not flattening (`-f` not given).

`-cl`
: This flag will cause geometry to be clipped at the (possibly bloated) grid cell boundaries. This applies whether flattening or not. Note that when not flattening, clipping does not guarantee that geometry is confined to the clip area.

`-e`[*N*]
: This will enable empty cell filtering, as described for the **Format Conversion** panel in 14.10. The options are:

  `-e` or `-e1`
  : Turn on both pre- and post-filtering.

  `-e2`
  : Turn on pre-filtering only.

  `-e3`
  : Turn on post-filtering only.

  `-e0`
  : Turn off all empty cell filtering (no operation).

`-p`
: This option specifies that an alternative "parallel" writing algorithm is used when creating output. In this case, the input file is read once only, and content is dispatched to the appropriate output files. The normal operation is sequential, where the input file is scanned for each output file. The parallel method is expected to be faster, though results may vary.

The command will create a temporary CHD, if necessary. Each grid region is written out sequentially, in the manner of windowing from the **Format Conversion** panel from the **Convert Menu**.

19.3 Current Directory

19.3.1 The !cd Command: Change Directory

Syntax: `!cd` [*directory*]

The **!cd** command changes the current working directory, as known to *Xic*, to *directory*. If no *directory* is given, the user's home directory is understood.

19.3.2 The !pwd Command: Print Directory

Syntax: `!pwd`

This command will print the *Xic* current working directory on the prompt line.

19.4 Diagnostics

19.4.1 The !time Command: Show Elapsed Time

Syntax: `!time`

Print the elapsed program run time, in seconds, in the console window.

19.4.2 The !timedbg Command: Show Internal Run Times

Syntax: `!timedbg` [`y|n` [-*level*] [*logfile*]]

This command enables or disables printing of internal timing information for display and DRC operations, and others.

If given with no arguments, a message is printed on the prompt line indicating whether or not timing info is being printed.

If the first argument is "`y`" or "`on`", timing information will be printed. This can be followed by an optional *level* which is an integer (following a hyphen) that sets the maximum level of sub-timing info to print. If 0, only the "top level" timing results are shown. If a file name appears, it gives a path to a file where the information will be written. Otherwise, or if the file can't be opened, output goes to the console window.

If the first argument is "`n`" or "`off`", timing information will not be printed. This has no effect unless timing info printing is enabled.

In the output, indentation is used to indicate the "level" of the measurement. Times printed for a given level include all of the times listed above at a greater indentation level after a previous line at the same level. A greater indentation level indicates a timing measurement of a sub-component of the operation.

19.4.3 The !xdepth Command: Show Transform Depth

Syntax: `!xdepth`

This prints two numbers on the console. The first number is the current transform stack depth, which should always be 0. The second number is the transform stack maximum depth used since the last **!xdepth** call or program start. This is rather useless except for debugging "Transform stack full" errors.

19.4.4 The !bincnt Command: Database Object Allocation

Syntax: `!bincnt` [*layername* [*level*]]

This is for debugging purposes, and for the curious.

This command prints some database statistics on the console window. If no *layername* is given, the layer examined will be "`$$`", the internal layer that contains subcell instances. The message will look something like

```
Cell noname Layer CSP
levels 3, nodes 7, frac 0.928571, items 46 (allocated 46)
```

This indicates that the tree structure for the data items on layer CSP has depth 3, 7 nodes other than the data nodes, occupancy fraction 0.93, and 46 data items, which matches the cached allocation number.

If a number follows the layer name, the enclosing bounding box for each sub-tree at the given level is transiently shown on-screen.

19.4.5 The !netxp Command: Check Net Expression

Syntax: `!netxp` *net expression*

This will parse the given net expression (as described in 4.2.8) into an internal representation, then reconvert this to a string which is printed in the console window. The expression will be iterated, and each bit expression will also be printed. This is a diagnostic for the net expression parser, but may also be useful to the user who is learning about net expressions.

19.4.6 The !pcdump Command: Dump Parameterized Cell Data

Syntax: `!pcdump` [*filename*]

This will dump the default parameter list for every parameterized cell (pcell) evaluated during the session. The list is in the format of the pc_params property, including constraints. This can be useful for viewing the parameters and constraints of OpenAccess pcells, as they lack a native super-master and thus the pc_params property.

The argument is the name of a file to create for output. If not given, output goes to the console window.

19.5 Design Rule Checking

19.5.1 The !showz Command: Show DRC Test Areas

Syntax: `!showz` [`y|n`]

The **!showz** command will turn on/off a transient display of the test areas used during DRC. This is for debugging, or for the curious. Given without an argument, the current show state is toggled.

19.5.2 The !errs Command: Regenerate DRC Error Highlighting

Syntax: `!errs`

This command will update the DRC error highlighting from an existing DRC error log file. The action is identical with that of the **Update Highlighting** button in the **DRC Menu**.

As it is redundant, this command may be removed in a future release.

19.5.3 The !errlayer Command: Create Error Polygons

Syntax: `!errlayer` *layer_name* [*prpty_num*]

This command will create polygons on *layer_name* corresponding to the error regions currently stored in the list of highlighted design rule errors. The layer will be created if it does not already exist, and will be cleared before updating (*be careful!*). All objects are created in the current cell. The second argument, if given, is an integer greater than 0 that is taken as a property number. Each created object will be given a property with this number, with the text being the error message for the error. If the argument is given but is not an unsigned integer larger than 0, no properties are stored.

This action is identical with that of the **Create Layer** button in the **DRC Menu**. As it is redundant, this command may be removed in a future release.

19.6 Electrical

19.6.1 The !calc Command: Calculate Parameter Expression

Syntax: `!calc` *expression*

This command started out as a debugging aid for the parameter handling code, but is actually pretty useful.

The *expression* is a math expression involving constants, parameter names, and the usual math operations and functions as provided for *WRspice* expressions. This is separate from the script expression parser, but rather similar in operation (the two may merge some day). The new expression handler accepts the a ? b : c construct, which is one difference.

Before evaluation, all parameter definitions in the electrical current cell are tabulated. This includes the param properties of the cell, and any `.param` lines found in labels on the SPTX layer. Parameters found can be used by name in the expression.

19.6.2 The !check Command: Database Consistency Check

Syntax: `!check`

This command will perform a consistency check of the electrical part of the current cell, and report any problems on the console screen. Additionally, all labels which are not associated with a device or other property will become selected. This command is for debugging purposes. These checks are also performed when a new cell is read into *Xic*, with error messages directed to the log file. If errors are found, in many cases they are repaired. Use the **!check** command a second time to verify if the condition still exists.

Messages may be added to the `read_`*xxx*`.log` file produced when input is read if repairs were made.

19.6.3 The !regen Command: Regenerate Labels

Syntax: `!regen`

The **regen** command will regenerate all missing property labels in the schematic. This is useful if a label was accidently deleted or otherwise lost due to some error.

19.6.4 The !devkeys Command: Print Device keys

Syntax: `!devkeys`

This will dump the current device key mapping table to the console window. The device keys are set in an internal table, which can be augmented or overridden by setting DeviceKeyV2 properties in the device library (`device.lib`) file.

19.7 Extraction

19.7.1 The !antenna Command: Check MOS Antenna Effect

Syntax: `!antenna` [*layer_name layer_min_ratio*]... [*min_ratio*]

In the design of CMOS circuits, design rules and guidelines often provide a limit on the area of a wire net connected to a MOS gate. During processing, the wire net can act as an "antenna" which accumulates charge, potentially damaging the thin MOS gate oxide. This command provides checking of antenna nets.

Note that this is part of the extraction system and not DRC. The DRC system presently does not maintain a sophisticated enough state to identify device contacts or follow wire nets.

The **!antenna** command utilizes the values of the technology file extraction keywords Antenna (in physical layer blocks) and AntennaTotal. These keywords provide values which are used as defaults, which can be overridden from the command line.

If given without arguments, the **!antenna** command will generate an argument list constructed from the defaults (if any). This is displayed in the prompt area, where it can be edited by the user. The run begins when the user presses the **Enter** key. If there are no defaults, or if an argument was given to the command, there is no prompt and the command runs immediately.

With no parameters given, the command will identify and print an entry for each wire net in the hierarchy of the current cell which connects to a MOS gate. The results go to a file, created in the current directory, named *cellname*`.antenna.log`, where `cellname` is the name of the current cell. The user is given a chance to view this file when the operation completes.

The parameters provide a "filtering" function, whereby only entries outside of the filter range are printed in the file. The filtering parameter is the ratio of wire net area to total gate area connected to the net. These ratios can apply to individual layers contained in the wire net, or the total wire net area. Only entries that exceed given parameters are printed in the log file.

For example,

```
!antenna POLY 20 M1 30 50
```

This will print wire nets where at least one of the following is true:

1. The ratio of POLY area to gate area exceeds 20.
2. The ratio of M1 area to gate area exceeds 30.
3. The ratio of total wire net area to gate area exceeds 50.

Thus, the log file will typically contain only those nets that exceed the guidelines.

These "bad" nets can be displayed in the **Select Path** mode of the **Path Selection Control** panel. After the **!antenna** command has been run, and/or with the log file in the current directory, pressing the **Load Antenna file** button or the **f** key will prompt for an antenna net number. This is the number in the log file that begins the report for each net.

The file will be accessed, and the corresponding wire net will be extracted and highlighted. The wire net is identified via the reference bounding box provided in the log file, on the same line as the net number.

19.7.2 The !netext Command: Batch Physical Net Extraction

Syntax: `!netext` *arguments...*

PRELIMINARY – This is the initial implementation of a new capability. Feedback and wish-lists from users is encouraged.

The **!netext** command performs identification and extraction of physical wire nets from a layout. There are a number of modes and features, but the final result is generally an OASIS file containing a top-level cell with the same name as the original top-level cell, which contains a subcell for every wire net. Each subcell contains all of the conductors that comprise the net, as if the original hierarchy were flat. This file can be used as a starting point for further analysis, such as parasitic extraction using a field solver.

The full operation is performed in three stages.

Stage 1

1. Create a Cell Hierarchy Digest (CHD) in memory for the input file, if necessary.
2. Divide the area of the top-level cell into a logical grid.
3. For each grid area, the CHD is used to read into memory a flat representation of the grid area, clipped to the grid.
4. The wire nets for this area are identified. This can take into account device structures and exclusion areas.
5. An OASIS file is written to disk, which contains a subcell for each net found. Up to four edge-mapping files are also produced, one each for the edges that are shared with another grid cell. These files map the parts of the edge which coincide with the edge of a conducting object.

At the end of Stage 1, the work area on disk contains a number of OASIS files, one for each grid cell, and associated edge mapping files.

Note that the grid areas are processed sequentially. On a computer with limited memory, the grid size should be "small" so as to not exhaust available memory, but even a modest computer can process very large files. Note also that in theory this stage could easily be accelerated by use of multiple computers. Stage 1 is the most compute-intensive part of the flow.

Stage 2

The second stage compares the two edge files for each shared grid boundary, and generates an equivalence file. The equivalence file maps between the nets that abut at grid boundaries. Once the edge files have been processed, the edge files are deleted.

Stage 3

In the final stage, the individual OASIS files for each grid cell are combined, using the equivalence file, into a single OASIS file. There are two ways that nets that extend across grid boundaries can be handled. The "easy" way is to simply copy all net cells from all grid areas into the output. For the nets that connect to other nets, choose a "primary" subnet (cell). In this cell, instantiate the other net cells to which the primary subnet connects.

The alternative is to actually copy the subnet cell geometry into the primary cell. This format is easier to work with, but requires more time and memory to construct.

When the output file is written, the equivalence file and the Stage 1 OASIS files are deleted, and the operation is complete.

If a net has an assigned name in the source file, (e.g., through a label or from a terminal) A NXNAME property will be given to the created net files. This is property number 7149, and the string is the net name. This is not presently used by *Xic*, but the net name may be useful to the user. Beware if using a grid: if flattening, the top net cell will contain the net name properties from all grid cells, so there may be duplicate or inconsistent name properties. If not flattening, the primary and all subcells should be checked, each property applies only to the corresponding grid location. In either case, conflicting names would need to be dealt with somehow.

Command Arguments

`-f` *filename*
: This mandatory argument specifies the input source for batch net extraction. the *filename* can be a path to a layout file in a supported format, the access name of a CHD in memory, or a path to a saved CHD file.

  The technology file in use must match the source file, with the extraction parameters and keywords properly set up.

`-c` *cellname*
: This provides the name of the top-level cell for extraction. If not given, the top-level cell used will be either the cell configured into the CHD source, if any, or the lowest-offset top-level cell found in the source layout file.

`-g` *gridsize*
: if the "`-w`" (windowing) option is not given, this sets the grid size, in microns. Use of a grid minimizes memory consumption for handling large designs. For smaller designs gridding may not be necessary, so this option can be skipped or given as 0. In this case the entire bounding box of the top-level cell is understood. The OASIS file is produced, but there are no edge files, and no Stage 2 or Stage 3 steps.

  The choice of a grid size is machine and layout dependent. The objective is to choose as large a grid as possible, without exceeding memory limits or causing excessive page-swapping. In general, some experimentation may be required to find the "best" grid size. A starting point of 400 microns may be reasonable.

`-v`
If given, via objects will be included in the netlist cells and files. Via layers are the layers with the Via keyword given in the technology file. The objects on these layers are clipped to the intersection areas of the two associated conductors.

`-v+`
This is similar to `-v`, but in addition the "check layers" (if any), clipped to the via object, will also be included in net cells and files. The check layers are the layers used in the optional layer expression supplied on the `Via` line. This expression must be "true" for a via object to actually represent a connection. With `-v` given, the included vias are those that pass the check criteria, but the check layers are not included. With `-v+`, the check layers will be included.

If the generated netlist file is read back into *Xic* and extraction run, the `-v+` option will allow the nets to be correctly re-extracted. If the check layers are missing, this may fail, and extraction would certainly fail if vias are not included at all.

`-vs`
When there is no windowing or gridding in use and this option is given, standard vias will be retained as they are rather than being converted to equivalent geometry. The net cells will contain the standard via placements from the net, from all hierarchy levels, as subcells. Presently, this requires that the standard via sub-masters not be included in the source layout file, i.e., they are created within *Xic* as the file is read.

`-w` *l*,*b*,*r*,*t*
If a window is given, a grid size should not be given and will be ignored. In this case, there is no grid, and the rectangular area given, as comma-separated dimensions in microns, is read into memory and processed as if it were a grid cell. The OASIS file is produced, but there are no edge files, and no Stage 2 or Stage 3 steps. If all values are 0, the effective area is the bounding box of the top-level cell, which is the default when no area or grid is given.

`-b` *basename*
This supplies a basename for the generated files. It can have a path prefix, which will cause the generated files to be written in the given directory, which must exist. If this argument is not given, the name of the top-level cell is used as the basename.

`-nf`
By default, in Stage 3 processing, the net cells will be flat. If this argument is given, subnets will appear is cell instances in the "primary" net cell.

`-nc`
This will turn off compression in OASIS output files. This is not a good idea, unless compression is not supported by the reader.

`-ne`
This turns off the part of the extraction that recognizes device structures, leaving only conductor grouping for connectivity determination. This may be fine for some applications, and avoids computation. In MOS circuits, for example, if the Active layer is assumed to be a conductor, then all FETs will be shorted, drain to source. However, using a Conductor Exclude directive for Poly on Active should fix this.

`-l`
If this is given, when the flat data are read into memory for processing, any existing layer filtering

is kept. Without this option, when `-ne` is not given, all layers are read since these may affect device recognition. When `-ne` is given, only Conductor and Via layers are read.

`-k`
: If given, all working files are retained. Without this option, edge files, etc. are deleted when no longer needed.

`-s1`
: If given, the operation will stop at the end of Stage 1.

`-s2`
: If given, the operation will stop at the end of Stage 2.

The grid cells are assigned x,y index numbers, according to position, with the 0,0 cell located in the lower left corner. The cells are traversed left to right by row, from bottom to top. Each net in a grid is assigned a number, which is the group number from extraction. All three numbers are non-negative, and the triplets represent a unique designation for a subnet. The net cells in the Stage 1 OASIS files files are names "*x_y_n*", i.e., the three numbers separated by underscores.

In the final OASIS file, the net cells are renamed `n1`, `n2`, ..., replacing the triples with an index number. If instantiation is used, the subnet cells that are not primary nets retain the original names. The primary subnet from among a group of connected subnets is the one that is lowest in "traversal order", which is the lowest group number in the first grid cell seen in a sweep left to right in the rows, ascending in y.

19.7.3 The !addcells Command: Add Missing Cells

Syntax: `!addcells`

This command adds "missing" instances to the current cell, in physical or electrical mode. An instance is "missing" if it is referenced in the opposite mode of the current cell, but does not appear in the current cell. Cells are not added if they are empty. The new instances are arrayed below existing objects. For example, suppose one creates a schematic consisting of several subcells from some library. One can then switch to physical mode and use this command to obtain the physical instances, which can then be moved into place. This avoids having to use the **place** command (in the side menu).

19.7.4 The !find Command: Find Devices

Syntax: `!find` [*devicename*[.*prefix*[.*index*]]]

This command will find and highlight devices in physical layout windows showing the current cell, and also highlight the corresponding device symbols in windows showing the schematic of current cell. It is basically a command line version of the device listing/highlighting feature of the **Show/Select Devices** panel from the **Device Selections** button in the **Extract Menu**.

The argument list consists of at most three fields, separated by periods. Missing fields are wildcards. The *devicename* is one of the names from a device block in the technology file. The *prefix* is from the `Prefix` line of the device block. The *indices* is a list of space or comma-separated integers, or hyphen-separated ranges of integers. The integers are the index values of the physical devices. If this field is not given, any index value will be highlighted, otherwise only the devices with an index that matches a value or falls in a range will be highlighted.

With no argument, any existing device highlighting will be erased.

If the first component is empty, or the keyword `all`, all devices known from the technology file are acted on. Thus, "`!find all`" or "`!find .`" will display all known devices. One can also give, for example, "`!find ..1`" which will show all devices with index 1.

19.7.5 The !ptrms Command: Default Terminal Locations

Syntax: `!ptrms l|t` [`r`]

Options can be space separated or grouped. At least one of `l`, `t` must be given. If `l` is given, the cell label markers will be moved to the default locations to the right of the parent cell. If `t` is given, all device terminals will be undefined and moved to the lower left of the parent cell. These actions can not be undone. If `r` is given, the operation is performed recursively on subcells. The characters `c`, `d` are equivalent to `l`, `t`. This command is used primarily for debugging purposes.

19.7.6 The !ushow Command: Show Unassociated Elements

Syntax: `!ushow` [*types*]

This command will highlight unassociated objects. These are objects in physical mode that have no identified electrical counterpart, and vice-versa.

The *types* argument is a word containing characters that indicate the object types to display:

`g` or `n`	groups/nodes
`d`	devices
`s` or `c`	subcells/subcircuits

If this argument is omitted, "`gds`" is the effective value, which will show all unassociated groups, devices, and subcircuits.

The command works in physical and electrical modes. Display windows will highlight the appropriate unassociated objects for the window's display mode.

The highlighting is removed on a deselect operation, with the menu button or otherwise. Mostly, the objects are simply selected, however objects such as physical devices use other highlighting methods.

19.7.7 The !fc Command: Control Capacitance Extraction Interface

Syntax: `!fc` *keyword* [*arg* ...]

This command is a prompt-line equivalent to some of the functionality of the capacitance extraction interface described in 16.17.1. This interface is also controlled from the **Cap Extraction** panel, which is produced by the **Extract C** button in the **Extract Menu**.

The first argument is a keyword, which must be present and must be one of those listed below. Additional arguments are specific to the keyword. The keywords perform an operation that is equivalent to pressing one of the buttons in the **Cap Extraction** panel.

`dump` [*filename*]

This will dump a unified list file using the name given in the argument, or the default name if no name is given. The default name is the name of the current cell with a "`.lst`" suffix.

This is equivalent to pressing the **Dump Unified List File** button in the **Run** page of the **Cap Extraction** panel. The format is compatible with the *FasterCap* program from FastFieldSolvers.com, and also the Whiteley Research version of *FastCap*, the latter requires use of the FcPanelTarget variable.

`run` [`-i` *infile*] [`-o` *outfile*] [`-r` *resultfile*]

If an *infile* is specified, that file will be taken as input to the capacitance extraction program, as if the **Run File** button in the **Run** page of the **Cap Extraction** panel was pressed and the *infile* specified in the text input area. Otherwise, the action is as if the **Run Extraction** button was pressed instead.

The *outfile* is the file used for standard output from the extraction program during the run. If not given, a temporary file will be used, and destroyed when the run completes, after copuying the content to the results file. If a name is provided, that file name will be used, and the file will not be destroyed.

If no name is given for the *resultfile*, a default name will be used. This file will contain input to and output from the extraction run.

19.7.8 The !fh Command: Control Inductance/Resistance Extraction Interface

Syntax: `!fh` *keyword* [*arg* ...]

This command is a prompt-line equivalent to some of the functionality of the inductance/resistance extraction interface described in 16.18.1. This interface is also controlled from the **LR Extraction** panel, which is produced by the **Extract LR** button in the **Extract Menu**.

The first argument is a keyword, which must be present and must be one of those listed below. Additional arguments are specific to the keyword. The keywords perform an operation that is equivalent to pressing one of the buttons in the **LR Extraction** panel.

`dump` [*filename*]

This will dump a *FastHenry* input file using the name given in the argument, or the default name if no name is given. The default name is the name of the current cell with a "`.inp`" suffix.

This is equivalent to pressing the **Dump FastHenry File** button in the **Run** page of the **LR Extraction** panel.

`run` [`-i` *infile*] [`-o` *outfile*] [`-r` *resultfile*]

If an *infile* is specified, that file will be taken as input to the inductance/resistance extraction program, as if the **Run File** button in the **Run** page of the **LR Extraction** panel was pressed and the *infile* specified in the text input area. Otherwise, the action is as if the **Run Extraction** button was pressed instead.

The *outfile* is the file used for standard output from the extraction program during the run. If not given, a temporary file will be used, and destroyed when the run completes, after copuying the content to the results file. If a name is provided, that file name will be used, and the file will not be destroyed.

If no name is given for the *resultfile*, a default name will be used. This file will contain input to and output from the extraction run.

19.8 Graphics

19.8.1 The !setcolor Command: Set Attribute Colors

Syntax: `!setcolor` *resourcename colorspec*

This command changes the attribute colors used within *Xic*. The *resourcename* is a color keyword or alias from the list of attribute colors (see A.8.3). The *colorspec* is the name of a color or RGB triple in the same format as used in the resource file. Changing the colors will in general not change appearance until the feature is redrawn.

19.8.2 The !display Command: Export Rendering

Syntax: `!display` *display_string win_id*

This command will render the current cell in a foreign X window. The X window id is passed as an integer in the second argument. The first argument is the X display string corresponding to the server in which the window is cached. The area to display is the same area currently defined for the main drawing window. See the corresponding `Display` script function for more information.

19.9 Grid

19.9.1 The !sg Command: Save Grid in Register

Syntax: `!sg [`*regnum*`]`

There is a set of eight registers that can hold grid parameters. Thus, grids can be saved and quickly restored. Whenever the grid is changed, for example with the **Set Grid** command in the **Main Window** sub-menu of the **Attributes Menu**, the previous grid is saved in register 0.

This will save the grid of the drawing window containing the pointer (or the main drawing window if the pointer is not in a drawing window) into register *regnum*. The *regnum* must be an integer 0–7, and is taken as 0 if not given.

The grid can be restored from a register with the **!rg** command.

19.9.2 The !rg Command: Set Grid From Register

Syntax: `!rg` [*regnumber*]

This will set the grid of the drawing window containing the pointer (or the main drawing window if the pointer is not in a drawing window) to the grid stored in *regnum*. The *regnum*, if given, is an integer 0–7. If not given, 0 is understood. A register that has not been saved will return a default grid style (1 micron, no snapping, dot grid). In addition, the grid storage register 0 takes the value of the previous grid.

The grid can be saved to a register with the **!sg** command.

19.10 Help

19.10.1 The !help Command: Help Interface

Syntax: `!help` *word*

This is a back-door to the help system. The *word* is a keyword expected to be found in the help database, or a path to a text, html, or image file to view, or a URL string to access on the internet. If no *word* is given, a default help topic is shown.

The command invocation is aliased to the question mark ('`?`') key.

Information on the help database is provided in C.3. All menu commands have a short name which is given in the "tooltip" which appears when the pointer is stationary over the command button for a second or two. The help database keyword is generally this name, prefixed with "`xic:`".

General URLs must have the protocol specifier given. For example, "`http://wrcad.com`" is correct, giving only "`wrcad.com`" will not work.

The "help mode", where pressing menu buttons brings up help topics, which is active when the help is accessed through the **Help Menu**, is not active when the **!help** command is used.

19.10.2 The !helpfont Command: Set Help Font

Syntax: `!helpfont` *fontfamily-size*

This specifies the default proportional font family used in HTML viewer (help) windows, and applies to Linux/macOS releases only. Under Microsoft Windows, this command does nothing. This is the font used to render most text in the help windows.

If no argument is given, the font reverts to the internal default.

The *fontfamily-size* is given as a face name, followed by white space, followed by the base pixel size. The internal default is "`Sans 9`".

This command has limited value, as the fonts are most conveniently set with the **Font Selection** panel available in the **Attributes Menu** and from the help windows.

19.10.3 The !helpfixed Command: Set Help Fixed Font

Syntax: `!helpfixed` *fontfamily-size*

This specifies the default fixed font family used in HTML viewer (help) windows, in Linux/macOS releases only. Under Microsoft Windows, this command does nothing. The fixed font is used to render typewriter and preformatted text.

If no argument is given, the font reverts to the internal default.

The *fontfamily-size* is given as a face name, followed by white space, followed by the base pixel size. The internal default is "`Monospace 9`".

This command has limited value, as the fonts are most conveniently set with the **Font Selection** panel available in the **Attributes Menu** and from the help windows.

19.10.4 The !helpreset Command: Clear Help Cache

Syntax: `!helpreset`

This will clear the internal topic cache used by the help system. The cache saves topic references as offsets into the help (`.hlp`) files, so that if the text of a help file is modified, the offsets are probably no longer valid. This function is useful when editing the text of a help file, while viewing the entry in *Xic*. Use this function when editing is complete, before reloading the topic into the viewer. Although the offset to the present topic does not change when editing, so that simply reloading would look fine, other topics in the file that come after the present topic would not display correctly if the offsets change.

19.11 Keyboard

19.11.1 The !kmap Command: Read Key Mapping File

Syntax: `!kmap` *mapfile*

This will read a key mapping file as produced from the **Key Map** button in the **Attributes Menu**. The key mapping feature allows non-standard keyboards to be used with *Xic* without loss of features.

This command allows a mapping to be applied at any time. Older *Xic* releases would automatically read a mapping file if found at startup. This is no longer true, map files must be read explicitly, either with this command, or with the `ReadKeymap` script funtion. The operation can be performed from a starcup script if the mapping is expected to always be applied.

If the *mapfile* is not rooted, it will be searched for in the current directory, the user's home directory, and along the library search path, in that order.

19.12 Layers

19.12.1 The !ltab Command: Modify Layer Table

Syntax:
`!ltab a`[`dd`] *layername ...*
`!ltab i`[`nsert`] *layername* [*index*]
`!ltab rem`[`ove`] *layername ...*
`!ltab ren`[`ame`] *oldname newname*

This command has multiple forms, corresponding to the keyword given as the first argument. Only the initial letters needed to identify the keyword are required. The manipulations available from this command can also be performed graphically with the **Layer Editor** from the **Attributes Menu**.

If the second word is recognized as "`add`", and the remaining tokens are valid layer names, layers are created (or extracted from the removed list) and added to the end of the layer table.

If the second word is recognized as "`insert`", and the token that follows is a valid layer name, the layer will be inserted into the layer table at a position given by the integer *index*. If the *index* is missing, negative, or larger than the number of layers in the table, the layer is appended to the table. If the index is zero, the layer will be inserted at the index of the current layer. Otherwise, the layer is inserted into the table at the position given by the index, with one being the first (topmost) position.

The "`remove`" form removes the listed layers from the layer table. Removed layers are saved, and can be reinserted if needed.

The "`rename`" form renames the layer named *oldname* to *newname*.

19.12.2 The !ltsort Command: Alphanumerically Sort Layer Table

Syntax: `!ltsort`

This command will sort the layers in the layer table into alphanumeric order. This may be useful when examining the layers from an unknown archive file when *Xic* is started without a technology file. This operation is not undoable.

19.12.3 The !exlayers Command: List layers by Applied Keywords

Syntax: `!exlayers`

This command will list in the console window layers in the current technology that have the following keywords set: `Conductor`, `Routing`, `GroundPlane`, `Contact`, `Via`, `Dielectric`, `Planarize`, `DarkField`.

19.13 Layout Editing

19.13.1 The !array Command: Manipulate Instance Arrays

Syntax: `!array -u`
`!array -d` [*nx1*[–*nx2*] , [*ny1*[–*ny2*]]
`!array -r` [`nx` [+]= *val*] [`ny` [+]= *val*] [`dx` [+]= *val*] [`dy` [+]= *val*]

This command manipulates instance arrays. There are three forms:

`!array -u`
This will "unarray" all selected arrays. The arrays are converted to individual instance placements, in the same location and orientation as the original array elements.

`!array -d` [*nx1*[–*nx2*] , [*ny1*[–*ny2*]]
This form will delete a rectangular region of array elements. The undeleted elements will be configured into a new collection of arrays or single instance placements.

The command operates on a selected instance array, the most recently selected if there is more than one.

If no arguments follow the option character, the user is asked to click on or drag over the array, to define two points. The two points are transformed back into the coordinate system of the instance master, and define a rectangular region in the array indices in that space. The elements corresponding to this rectangle are deleted, and new arrays or separate instances are created to replace the undeleted elements.

Otherwise, the range of x and y indices to delete is given on the command line. These indices are non-negative 0-based, and the x and y ranges are separated by a comma. A range can be a single number, or two numbers separated by '–'. If a single number, the range is taken as that number only.

In the untransformed array, the 0,0 location is the lower-left corner.

Example:
Suppose that a 3x3 array is selected.
Erase the middle element: `!array -d 1,1`
Erase the rightmost column: `!array -d 2,0-2`

`!array -r` [`nx` [+]= *val*] [`ny` [+]= *val*] [`dx` [+]= *val*] [`dy` [+]= *val*]

This will reconfigure the array parameters of the first selected instance. It can convert instances into arrays and vice-versa.

All of the parameter groups are optional, but at least one group should be given or the operation does nothing. Each is in the form *keyword* [+]= *value*. It a '+' appears ahead of the '=', the *value* will be added to the existing value, otherwise the *value* is assigned. White space around '=' or '+=' is optional. The `nx` and `ny` are the number of columns and rows in the untransformed array. These integer values must be one or larger. The `dx` and `dy` are the array cell spacing in the untransformed x and y directions, given in microns.

Examples:
Add a column to the selected array: `!array -r nx+=1`
Add 1.5um additional space between elements: `!array -r dx+=1.5 dy+=1.5`

19.13.2 The !layer Command: Generate Layers

Syntax: `!layer [join|split|splitv] [-j | -s[h] | -sv] [-d` *depth* `| -da] [-r] [-c] [-m] [-f]` *layer_name* `[=]` [*expression*]

This command produces new geometry on a new or existing layer, by applying a layer expression which takes as input geometry from the same or other layers, from the current cell or from other cells in memory. The **Layer Expression** button in the **Edit Menu** provides a panel which duplicates the functionality of this command.

This new geometry can appear as an assemblage of trapezoids if either of the `split` or `splitv` keywords is given, or alternatively as a minimal number of complex polygons if the `join` keyword is given instead. If `splitv` is given, a vertical orientation is favored for the decomposition, whereas similarly `split` will produce a decomposition favoring a horizontal orientation. The default is the joined form if none of these optional keywords is given, except when simply copying from another layer in which case the default is to copy objects without change. The keyword "`splith`" is a synonym for "`split`". The options `-j`, `-s` or `-sh`, and `-sv` are equivalent to giving the `join`, `split`, and `splitv` keywords.

The **!layer** command, when using boolean operations, uses gridding to improve efficiency for large data sets. Internally, a square grid with origin at the lower-left corner of the cell bounding box is logically defined. The calculations are performed for each grid square that overlaps the cell area, and the results are combined. This can be more efficient that calculating the whole cell in one shot.

The default grid size is 100 microns square, which can be changed with the PartitionSize variable. This can be set to an alternate grid size in microns, as a floating-point number. The cell lower left corner is on the grid boundary. The operations are performed piecewise in each grid area that intersects the cell.

If this variable is set to "0", no grid is used, and operations will be performed over the entire cell at once.

The PartitionSize variable can be set with a control in the **Evaluate Layer Expression** panel from the **Layer Expression** button in the **Edit Menu**, or with the **!set** command.

When joining objects, there are several variables which fine-tune the operation. See the description of the **!join** command (19.13.13) for information.

If *layer_name* does not exist in the layer table, it will be created. Otherwise, the *layer_name* is the short or long name of an existing layer. If a new layer is created, its name is generated from the given name in the same way as in the technology file layer definitions.

The *expression*, if given, involves layer names and operators as in the DRC layer expressions (see 15.1). The result of the expression is created on *layer_name*. Thus, this command provides a means of creating a new layer from geometry on existing layers. It operates on the physical part of the current cell. Labels are ignored. The same *layer_name* can exist on both sides of the expression, in which case the contents of the *layer_name* is replaced with the result of *expression*. The equal sign between *layer_name* and *expression* is optional.

If no *expression* is given, the new layer will be created if necessary, which will be the only effect if done. If the *layer_name* already exists, and one of the `split`, `splitv`, or `join` keywords is given, the operation will be applied to that layer, much like the **!split** and **!join** commands.

If the *expression* consists of a layer name only, the objects on that layer will be copied to *layer_name*, and split/joined if the keywords are given. When simply copying and/or joining/splitting, no grid partitioning is used.

Copying and splitting/joining are available in electrical mode. Other operations require running the **!layer** command in physical mode, and apply to physical data.

There are several option flags which can be given. These must appear before *layer_name* in the command line. The options can be given separately as shown in the syntax example above, or grouped, e.g., "`-dmf` *depth*" is equivalent to "`-d` *depth* `-m -f`". Any combination of grouped or single flags can be used. If a group contains '`d`', the token that follows must be the *depth*.

`-j`
: Equivalent to giving the `join` keyword.

`-s` or `-sh`
: Equivalent to giving the `split` keyword.

`-sv`
: Equivalent to giving the `splitv` keyword.

`-d` *depth*, `-da`
: The *depth* is a non-negative integer indicating the depth into the cell hierarchy to process. It can also be a word starting with the letter '`a`' to indicate all levels. If 0 (the default) only objects in the current cell are processed. If "`all`", all objects in the hierarchy may be used to generate the new objects, effectively flattening. The `-da` variation is equivalent to "`-d all`".

`-r`

This applies when the *depth* is larger than 0. When given, the *expression* is evaluated in all cells in the hierarchy to *depth*, using only objects in that cell and creating objects in that cell. This is very different from the behavior without this flag given, which is to create all objects in the current cell.

`-c`

By default, *layer_name* is cleared before the *expression* is evaluated, so that the layer contains only the result of the operation on command completion. If this flag is given, the layer will not be cleared, so that the original objects will be retained on the layer.

`-m`

When this flag is set, objects added to *layer_name* will be merged with existing objects, using the same merging as established with the **Merge new boxes and polys with existing boxes/polys** and **Clip and merge new boxes only, not polys** check boxes in the **Editing Setup** panel from the **Edit Menu**, or the corresponding variables. Use of full polygon merging can greatly increase processing time, simple box clipping/merging has much lower overhead. Merging may reduce the object count in the layout.

The merging will defeat the purpose of the split keywords, so the user should consider whether merging is appropriate. Merging includes the initial objects on the *layer_name* if it is not cleared, and the accumulated objects as evaluation takes place.

`-f`

This flag indicates "fast" mode, where undo list generation and any merging (other than a join operation) are skipped. This operation is not undoable, so this option should be used with care. It speeds processing and reduces memory use.

The user will be prompted to confirm before the operation is actually initiated.

Examples

Clear layer M0:
`!layer M0 0`

Copy M1 to layer NEW:
`!layer NEW M1`

Copy the inverse of layer M1 to layer NEW:
`!layer NEW !M1`

Copy the intersection areas of I1 and I2 to NEW:
`!layer NEW I1&I2`

Copy the R1 and R2 areas to NEW:
`!layer NEW R1|R2`

Extended Layer Names

The layer names in layer expressions in the **!layer** command can actually be given in an extended form:

lname[`.`*stname*][`.`*cellname*]

Most generally, the "layer" name consists of three tokens, two of which are optional (indicated by square brackets above). The tokens are separated by a period ('`.`') character. The individual tokens can be double-quoted (i.e., using the double-quote ('`"`') character), which must be used if the tokens contain non-alphanumeric characters. The period separators must appear outside the scope of any quoting.

lname
This is a short or long layer name, as found in the layer table.

stname
The name of a symbol table which contains the *cellname*.

cellname
The name of a cell.

If only one separator appears, the token that follows is taken as the *cellname*, and the current symbol table (see 9.3) is assumed.

The *cellname* is the name of a cell used as the source for geometry. If no *cellname* is given, the name of the current cell is understood. The odd case of an empty *stname* indicates the "`main`" symbol table, e.g., `layer..cell` is equivalent to `layer.main.cell`.

If the *cellname* starts with the " character, and no symbol table name is given, then the rest of the *cellname* is taken as the name of a "special" database, as created with script functions like `ChdOpenZdb`. If found, geometry will be obtained from the database rather than a cell. Otherwise, when a *cellname* is given, the geometry is obtained from the given cell, as if it were overlaid on the current cell. The *cellname* (or any of the three tokens) can be double quoted, and must be quoted if the name contains a '`.`' character, for example `CPG."mycell.xic"`.

If a *stname* is given, and the name matches an existing symbol table name, the cell is obtained from that symbol table. If the symbol table name is given, the *cellname* field must appear, but can be empty (a trailing period) which indicates the name of the current cell.

If the *stname* is given, and the cell is not in this table, it will be opened from disk into the given table (not the current table) if found as a native cell file in the search path.

The coordinate origin of the source cell is taken as the origin of the current cell. The source cell must be in memory, or be in a native cell in the search path.

Objects read from a "special" database are clipped to the boundary of the cell being added to. No such clipping is done when objects are read from another cell.

Advanced Examples

Suppose one has two versions of a cell, `cell` and `cell_old`, and one needs to know if they differ on layer `M1`. Open a dummy cell for editing, then issue

```
!layer ZZ = M1.cell^M1.cell_old
```

Press the **Home** key to view the entire cell space. Any geometry shown on the new dummy layer `ZZ` is the exclusive-OR of the geometry on `M1` of the two cells, i.e., the difference. If there is no geometry on `ZZ`, `M1` is the same in `cell` and `cell_old`.

As a variation, suppose that the user has done the following:

Set symbol table to ``old''.
open oldstuff/mycell
Return to previous symbol table.
open newstuff/mycell

There are two versions of mycell in memory. To compare the layer M1 in the two cells, one could then enter

```
!layer ZZ M1^M1.old.
```

Then the ZZ layer, which consists of the exclusive-OR of old and new M1 in mycell, would be added to the current mycell. Pressing the **Tab** key undoes the addition.

Suppose one wants to import the inverse of the geometry on layer VIA from cell into the current cell, also on layer VIA:

```
!layer VIA = !VIA.cell
```

The VIA layer now consists of the inverse from cell. Any geometry that existed on VIA in the current cell before the command was given is deleted. The bounding box of the current cell may have been expanded to include the bounding box of cell. The area used to create an inversion is the rectangle bounding all cells referenced in the expression, plus the current cell.

Suppose one simply wants to copy the geometry from layer M2 of cell into the current cell:

```
!layer M2 = M2.cell
```

The M2 layer now consists of the geometry on M2 from cell. The bounding box of the current cell may have been expanded, in which case some of the M2 features may be off-screen (press the **Home** key to view the entire cell). Any objects previously existing on M2 in the current cell are deleted before the operation.

19.13.3 The !mo Command: Move Objects

Syntax: !mo *x* [*y* [*layer_name*]]

The **!mo** command will move selected objects to a new location offset by *x*, *y* (in microns) from the original object. If not given, *y* is zero.

The third argument, if given, will allow a layer change during the move. It should be the name of a layer that is not the current layer. How this is applied depends on the setting of the LayerChangeMode variable, or equivalently the settings of the **Layer Change Mode** pop-up from the **Set Layer Chg Mode** button in the **Modify Menu**. For the layer change, the passed *layer_name* is taken as the "new current layer", however the actual current layer does not change. Subcells are moved without regard to *layer_name* or the layer change mode.

There is a companion **!co** (copy) command.

19.13.4 The !co Command: Copy Objects

Syntax: !co *dx* [*dy* [[-l] *layer_name*] [[-r] *rep_count*]]

The **!co** command will copy selected objects to new locations. The *dx* and *dy* are translation values in microns. If *dy* is not given, it is taken as 0. A *dy* value must be given if additional arguments are given.

There are two additional arguments than can appear: a replication count, and a layer name. An integer value that is not identical to a layer name is taken as a replication count, otherwise a layer name is assumed. The optional flags "`-l`" and "`-r`" can appear ahead of the token to enforce the interpretation.

The replication count specifies how many copies, spaced by *dx*,*dy*, are generated. For example, if the count is 2, new objects would be created at offset *dx*, *dy*, and 2**dx*,2**dy*. If not given, or the value is not in the range 1–100000, only one copy is made.

The *layer_name* argument, if given, will allow a layer change during the copy. It should be the name of a layer that is not the current layer. How this is applied depends on the setting of the LayerChangeMode variable, or equivalently the settings of the **Layer Change Mode** pop-up from the **Set Layer Chg Mode** button in the **Modify Menu**. For the layer change, the passed *layer_name* is taken as the "new current layer", however the actual current layer does not change. Subcells are copied without regard to *layer_name* or the layer change mode.

There is also a companion **!mo** (move) command.

19.13.5 The !spin Command: Rotate Objects

Syntax: `!spin` *x y angle* [*layer_name*]

This command will rotate all selected objects about *x*,*y* (given in microns) by *angle* (given in degrees) counter-clockwise. The functionality is similar to the **spin** command in the side menu.

Subcells and labels will be rotated in increments of 45 degrees in physical mode, 90 degrees in electrical mode, to the closest angle to that given. Other objects can be rotated by any angle.

The *layer_name* argument, if given, will allow a layer change during the rotation. It should be the name of a layer that is not the current layer. How this is applied depends on the setting of the LayerChangeMode variable, or equivalently the settings of the **Layer Change Mode** pop-up from the **Set Layer Chg Mode** button in the **Modify Menu**. For the layer change, the passed *layer_name* is taken as the "new current layer", however the actual current layer does not change. Subcells are rotated without regard to *layer_name* or the layer change mode.

19.13.6 The !rename Command: Rename Cells

Syntax: `!rename` [*prefix*] [[-s] *suffix*]

The purpose of the **!rename** command is to allow modification of all of the cell names in a hierarchy. In *Xic*, every cell name in the symbol table must be unique. When combining designs from various sources, it is necessary to take measures to avoid name clashes. The **!rename** command allows the manipulation of prefixes/suffixes of all of the cell names in a hierarchy. For example, each cell name can be prepended with a unique prefix, say the author's initials.

The *prefix* and *suffix* are string tokens. If two string tokens are given, the "`-s`", which implies suffix, can be skipped. The string tokens can contain any alphanumeric characters plus '$', '?', '_'. String tokens given in this form will be prepended/appended to the current cell name, and each cell name used in the hierarchy. The string tokens can also have the form /*str*/*sub*/ which indicates a substitution. This causes the *str* if it appears as a prefix/suffix of a cell name to be replaced by *sub*. The *sub* can be empty (i.e., the form is /*str*//) which can be used to undo the previous addition of a prefix or suffix. Forms like //*sub*/ are equivalent to just giving *sub* as a string.

19.13.7 The !svq Command: Save Selections in Register

Syntax: `!svq` [*regnum*]

This will save the current selections into a "register" which can be recalled later with the **!rcq** command. There are ten registers corresponding to given digits 0-9, or if no number is given 0 is understood.

The registers are actually just dummy cells in memory, which will appear in listings as "`$$$$REG0`" through "`$$$$REG9`". These should not be edited directly or instantiated.

19.13.8 The !rcq Command: Recall Selections from Register

Syntax: `!rcq` [*regnum*]

This will recall the contents of the register whose index 0–9 is given, attaching the objects to the mouse pointer where they can be placed by clicking in an active drawing window. The register must have been defined previously with the **!svq** command. If no number is given, 0 is understood.

19.13.9 The !box2poly Command: Object Type Conversion

Syntax: `!box2poly`

This command converts selected boxes to polygons in the database. The command is not expected to be useful except for debugging purposes. The box database entry uses less space than that of a single polygon.

19.13.10 The !path2poly Command: Outline to Polygon Conversion

Syntax: `!path2poly`

This will convert selected wires to polygons representing the wire path. The first and last vertex of the wire must be the same. The width and end style of the wire are ignored. The polygon represents the internal area specified by the path vertices.

19.13.11 The !poly2path Command: Polygon to Outline Conversion

Syntax: `!poly2path`

This will convert each selected polygon to a wire, using the same path as the polygon boundary, and the same layer as the polygon. The wire width will be the default width for wires on the layer. The end style of the wire will always be "flush ends", the default wire end style for the layer will be ignored.

19.13.12 The !bloat Command: Expand Objects

Syntax: `!bloat` *dimen* [*mode*]

The *dimen* is a dimension in microns. The command will operate on selected objects, and alter the dimensions according to the *dimen* given. If the *dimen* is positive, the parts of edges that do not contact or overlap with a selected object on the same layer will be pushed out by *dimen*, expanding the objects. If *dimen* is negative, the reverse occurs: objects will shrink, but adjacent objects will remain touching. Objects may be severed into two or more pieces if the *dimen* is negative, or may disappear entirely.

Only boxes, wires and polygons are affected. Wires and and possibly boxes become polygons after the operation. An object is deselected if it is modified.

There are a number of operational details and choices available with the *mode* integer, whose bits represent flags. This value can be given as a decimal integer, or as a hexadecimal number following "`0x`". If the *mode* argument is missing, a value of 0 is implied.

bits 0-1 (0x1, 0x2)
The two LSBs specify the basic algorithm mode, as described below.

bit 2 (0x4)
When set, the algorithm mode calls the "old" bloating algorithms, as used in releases prior to 2.5.67. If this bit is set, all of the other flag bits are ignored.

bit 3 (0x8)
When set, the return is the edge template, and no bloating is done. The edge template is a collection of polygons that cover the edges of objects that would be bloated, as a path, whose width is twice the *dimen*. When bloating, the edge template is either added to the objects being bloated, or clipped from them, depending on the sign of *dimen*.

bits 4-7 (0x70)
These three bits specify the corner "fill-in" mode, used when constructing the edge template. Consider a vertex and two adjacent edges. Imagine the rectangles formed from these edges by constructing parallel edges plus and minus *dimen* perpendicular to the edges, and using the four endpoints of the parallel segments to define two rectangles. The two rectangles will overlap, with a notch at the original vertex location. Adding a suitable shape to fill in this notch, thus creating a smooth transition, is the purpose of the corner fill-in.

The corner fill-in shape has three points initially defined, the vertex, and the two projections along the ends of the constructed rectangles. The differences between the fill-in modes is where (or if) we add the fourth point to the fill-in polygon. The choices are as follows:

bits 4-6: 000 ("clip" mode)
The angle is bisected, and the point added is a distance given by the absolute value of *dimen* from the vertex, along the bisector. This produces a rounding effect at the corner.

bits 4-6: 001 ("flat" mode)
No fourth point is added, only a triangle formed by the existing three points is used.

bits 4-6: 010 ("extend" mode)
The point added is the projected intersection of the outer edges of the two rectangles. For acute angles, the distance to the extended vertex is unconstrained.

bits 4-6: 011 ("extend-1" mode)
The point added is the projected intersection of the outer edges of the two rectangles. For acute angles, if the corner would extend too far, is is clipped (similar to the "clip" mode).

bits 4-6: 100 ("extend-2" mode)
This mode is similar to the "extend-1" mode, but provides a different and more aggressive clipping of acute angles.

bits 4-6: 101 (unused)
This code is reserved for expansion, produces no corner fill.

bits 4-6: 110 (unused)
This code is reserved for expansion, produces no corner fill.

bits 4-6: 111 (no fill)
This produces no corner fill.

Small angles will use the "flat" corner fill mode to avoid adding unnecessary vertices, in all modes.

bit 7 (0x80)
When using the "extend" corner modes, it is possible in certain geometries that the extended corner will occur on the opposite side of an edge rectangle from some other edge, which will produce unexpected features in the bloating result. In order to prevent this, a rather expensive test is performed. Setting this bit will skip the test, speeding up the operation somewhat. In Manhattan geometry, this test can always be skipped.

bit 8 (0x100)
Internally, the grouping operation that is part of the preparation for the edge template generation is skipped. This is an internal artifact, and this flag should not be set. However, if only a single object is being bloated, this flag may provide a slight speed improvement.

bit 9 (0x200)
Internally, clipping/merging of the trapezoid list passed to the bloating function is skipped. This is an internal artifact and this flag should not be set.

bit 10 (0x400)
When this bit is set, a scaling algorithm is applied during the bloating, which very slightly (+/- one internal unit) affects output coordinates. This is the result of a very specialized customer request that output exactly match that from another tool, and is not likely to be generally useful.

The scale fix will provide more accurate bloating when all angles are multiples of 45 degrees. It is not needed for Manhattan geometry, and for angles other than 45 degree multiples, it can actually reduce accuracy. For best accuracy in the all-angle case, the DatabaseResolution variable can be set to a larger value.

bit 11 (0x800)
When this bit is set, the trapezoid collection used to define the edge template will not be clipped and merged before use. This is an internal artifact and this flag should not be set.

bit 12 (0x1000)
When this bit is set, the resulting trapezoid collection produced for the edge template or by the bloating operation will not be joined into polygons.

The basic algorithm for modes 0-2 works as follows:

1. The collection of objects to bloat is converted to a trapezoid representation.
2. The resulting trapezoid list is grouped into multiple lists of spatially disjoint lists, where each list is mutually connected and no trapezoid touches or overlaps a trapezoid from another list.
3. For each list, the line segments representing the trapezoid edges are tabulated.

4. The edge list is clipped against itself to remove mutually overlapping regions. The remaining edges are the "external" edges, where one side is area outside of the trapezoid group.

5. Each edge is converted to a rectangle that covers the edge and extends $+/-$ the bloat width normal from the edge (note that these rectangles are rotated by an arbitrary angle, depending on the angle of the line segment).

6. The rectangles are converted to trapezoids. The non-Manhattan rotated rectangles are represented by three trapezoids.

7. A polygon, implemented as trapezoids, is added at each vertex, to fill in the transition between edge segments. The list of all these trapezoids represents a path along the external edges of the original trapezoid group.

8. If the bloat value is positive, the edge list is or'ed with the original trapezoid list. If the bloat value is negative, the edge list is clipped from the original trapezoid group. If bit 3 is set, this step is skipped, and the edge list is passed to the next step.

9. The resulting trapezoid list is merged into polygons, representing the operation result.

bloat mode 0

If a trapezoid group is entirely Manhattan, meaning that all edges are horizontal or vertical, no corner vertex fill-in takes place. Instead, the vertical line segments are extended by the (positive) bloat dimension. Thus, bloated Manhattan objects always remain Manhattan.

Otherwise, the polygon to fill the empty area at a vertex between the segment rectangles is computed, according to the corner fill-in mode. This may add vertices to the resulting figures, giving rounded corners.

bloat mode 1

This mode is faster, but is not recommended for non-Manhattan geometry. The vertical segment ends are extended by the bloat dimension to cover (assumed) Manhattan corners. Non-Manhattan segments are added as a single trapezoid with a width computed from the bloat dimension. Note that this can cause small protrusions and other anomalies to appear after bloating.

bloat mode 2

This is the same as bloat mode 0, however the corners of Manhattan and non-Manhattan objects will be treated the same. The corners of positive-bloated boxes may be rounded, unlike mode 0.

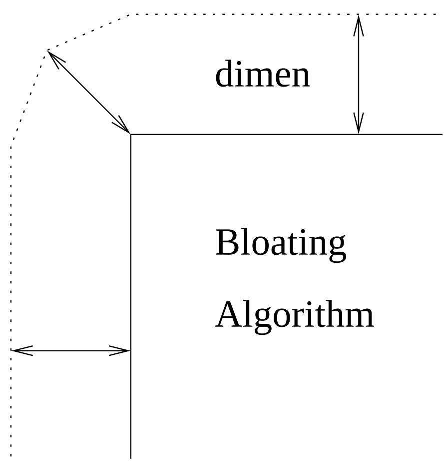
bloat mode 3

This mode uses the DRC sizing functions to perform the bloating operation, with results similar to mode 2. All of the other flags except for bit 3 (edge template) are ignored with this choice. If bit 3 is set, an edge template is created, extending out of the original figure if the bloat value is positive, or inward if the bloat value is negative.

This mode works best if a **!join** is performed before the bloat. This algorithm is rather compute intensive and slower than the other algorithms. In this algorithm, parts of edges that touch an object on the same layer will not be moved, whether or not the adjacent object was selected. In the other algorithms, unselected objects are completely ignored.

Presently, if bit 2 is set, the "old" algorithms will be used. These give results similar to the new algorithms, but are slower.

Figure 19.1: The default algorithm used in the **!bloat** command to enlarge an object.



old mode 0

In the description, we assume that the object is being expanded, i.e., the *dimen* is greater than zero. For each edge, an extension out of the object normal to the edge is created. For each corner where the edge projections do not overlap, a 4-sided polygon is created. Three of the vertices are the figure corner vertex and the ends of the two adjacent projections. The fourth vertex is placed along the bisector of the angle formed by the other three vertices, a distance *dimen* from the object corner vertex. All of the projections are joined to the original object to create the expanded object. Note that the corners become rounded, i.e., bloated rectangles become polygons. Figure 19.1 illustrates the algorithm.

If the *dimen* is less than zero, the object will be shrunk. In this case, the projections extend into the object, and the new object is formed by clipping these regions from the object.

old mode 1

This algorithm works with a trapezoid decomposition of the objects to be modified. An expansion is very fast, but a shrink requires polarity inversion of the trapezoid list, so is somewhat slower. This algorithm is not really recommended for non-Manhattan geometry, since in working at the trapezoid level without considering adjacency, small artifacts are often introduced at non-Manhattan corners.

The algorithm takes the following steps:

If $dimen > 0$ (expanding):

1. Decompose all selected objects on a given layer into a trapezoid list.
2. Create a second list containing trapezoids derived from the edges of trapezoids in the first list, created to enclose each edge and the surrounding area to $+/-$ *dimen* normal to the edge.
3. Merge the two lists and join into polygons.

If $dimen < 0$ (shrinking):

1. Decompose all selected objects on a given layer into a trapezoid list.
2. Invert the list in a rectangle that encloses all trapezoids bloated by *dimen*.

3. Create an edge trapezoid list from the inverted list.
4. Clip out the regions of the original list that overlap trapezoids in the edge list.
5. Merge the resulting list into polygons.

old mode 2
In this algorithm, for *dimen* larger than 0, the objects are first joined into maximal polygons, i.e., no two of these polygons abut or overlap. The vertex list of each polygon is used to construct a "wire" of width $2 * dimen$, which is then converted to a polygon representation. The wire polygon covers the edge of the original polygon, extending by *dimen* inside and outside of the figure. Each polygon becomes the union of the original polygon and its "wire" polygon. If *dimen* is less than zero, the geometry is inverted first as in the previous algorithm. Thus, the edge "wires" around the clear areas are found. These are clipped from the dark areas, yielding the final figures. Without the inversion, polygons with holes would not be processed correctly.

Note that bloating modes 1 and 2 will not round the corners, i.e., Manhattan corners remain Manhattan.

19.13.13 The !join Command: Join Touching Objects

Syntax: `!join` [`-l | -a`]

This command will merge boxes, polygons, and optionally wires into complex polygons. Use of merged geometry can reduce memory use and the size of the layout data file.

The **Join**, **Join Lyr**, and **Join All** buttons in the **Join or Split Objects** panel from the **Join/Split** button in the **Edit Menu** provide an equivalent to the **!join** command.

There are three basic operating modes. The **!join** command without arguments will join only selected objects. With the "layer" argument, all objects on the current layer may be joined, With the "all" argument, objects on any layer may be joined.' In these two cases, objects will be joined whether selected or not. For the arguments, the traditional "`-`" is actually optional, and only the first letter is considered, case insensitive, So, "`!join -a`", "`!join All`", and "`!join apple`" are all equivalent.

If a layer has the `NoMerge` keyword applied, in general joining (merging) is forbidden on the layer. However, this is overridden by the **!join** command without arguments. In this mode, the user must select the objects to join, and it is assumed that the user really wants them joined. In the other modes, objects on layers with this keyword set will **not** be joined. The user must first remove the keyword with the **Tech Parameter Editor** from the **Attributes Menu**, or otherwise.

In any case, the layer must be visible. With the "all" option, the layer must also be selectable.

The **!join** command, the **Join**, **Join Lyr**, and **Join All** buttons, the `Join`, `JoinObjects` and `GroupObjects` script functions, and other commands such as **!layer** which perform a join operation, are sensitive to four variables which fine-tune the behavior and performance. The default values emphasize speed but limit the complexity of resulting polygons. The user may need to set one or more of these variables in order for the operation to meet requirements. These variables can be set from the **Join or Split Objects** panel, using the analogous controls.

In addition, the **JoinSplitWires** variable, which also has a corresponding check box in the **Join or Split Objects** panel, determines whether wires are included in join operations. By default, wires do not participate in the join, however if the variable (or equivalently, the check box) is set, wires will behave the same as polygons.

To join a set of objects, the first step internally is to decompose each object onto a collection of trapezoids. As the objects are decomposed, the trapezoids are added to a list, which will be sent on to the function which performs the join. The variable JoinMaxPolyQueue sets the limit on the number of trapezoids that can accumulate before the list is processed. All or none of the trapezoids from a given object are added to the list, i.e., objects are not broken up at this point. If the addition of the trapezoids would cause the list to exceed the limit, then the list is sent on for processing, and a new list started. If JoinMaxPolyQueue is set to 0, there is no limit, and only a single list will be processed. When this variable is not set, the effective default value is 0 (no limit).

When a list is sent on for processing, the first operation is to break up the list into groups. Each group contains one or more trapezoids, such that the trapezoids in each group are "connected", i.e., the aggregate forms a single figure. The variable JoinMaxPolyGroup specifies a limit on the number of trapezoids in any single group. If this limit is reached, no additional trapezoids are added, instead they are placed in a new group or possibly some other existing group. If this variable is set to 0, then no limit is applied, and in this case all groups are guaranteed to be disjoint. When this variable is not set, the effective default value is 0 (no limit).

For each group, one or more polygons are created, which exactly cover the area of the trapezoids. The variable JoinMaxPolyVerts specifies a limit on the number of vertices which can appear in any single polygon. Thus, if the limit is reached, more than one polygon will be generated. If this variable is set to 0, then no limit is applied, and a single polygon will be created for each group. When this variable is not set, the effective default value is 600.

When the effective value of JoinMaxPolyVerts is nonzero, the JoinBreakClean variable determines now the partitioning is done. If this variable is not set, then the polygons are built up by adding trapezoids until the vertex limit is reached, at which point a new polygon is started, and constructed using the remaining trapezoids. The process continues until all trapezoids have been included in a polygon. The resulting collection of polygons may have complicated boundaries that interleave in a rather random way.

If JoinBreakClean is set, the vertex limit is initially ignored, and a single polygon is created from all of the trapezoids. If the vertex limit is exceeded, the polygon is split in two pieces, either horizontally or vertically. If either piece still exceeds the limit, it is subdivided in the same way, and so on until all polygons are within the limit. In this case, the boundaries are Manhattan. This processing is more compute-intensive than the other approach, but provides a better looking layout.

19.13.14 The !jw Command: Join Wires

Syntax: `!jw` [`-l`]

Without arguments, this command will take the most recently selected wire, and recursively join it with other similar (same width and layer) wires that share an end point.

If the `-l` option is given, all wires on the current layer in the current cell will be joined with any similar wires that share an endpoint.

The command works in electrical and physical modes. Its initial purpose was to fix designs imported from another EDA tool that had all wires as two-vertex segments. Within *Xic*, an attempt is made to keep wires maximally joined in general, which is more efficient, so this command is probably rarely needed.

19.13.15 The !split Command: Atomize Objects

Syntax: `!split` [v|V|1]

This is basically the reverse of **!join**. Selected polygons will be converted to collections of boxes and four-sided polygons.

However, objects on layers with the `NoMerge` keyword applied cannot be split (or joined). The **Edit Tech Params** button in the **Attributes Menu** brings up an editor that allows changing of this status.

This functionality is also available from the **Split Horiz** and **Split Vert** buttons in the **Join or Split Objects** panel from the **Join/Split** button in the **Edit Menu**.

Wire objects can be split similar to polygons if the **Include wires (as polygons) in join/split** check box in the **Join or Split Objects** panel is set, or equivalently if the JoinSplitWires variable is set.

If an argument is given that has `v`,`V`, or `1` as a first character, the splitting orientation is along the vertical, i.e., objects are divided by vertical lines that intersect the vertices. This is the mode used by the **Split Vert** button. Otherwise, splitting favors the horizontal orientation.

19.13.16 The !manh Command: Convert to Manhattan Polygons

Syntax: `!manh` *min_box_size* [*mode*]

This command applies to selected polygons. It will convert each polygon to a Manhattan approximation, meaning that all sides will be horizontal or vertical.

The first argument is the size, in microns, of the minimum box width/height used to approximate non-Manhattan parts of the polygon.

The second argument is an integer that provides a choice of algorithms. If this argument is not given, a zero value is understood. Presently, there are two Manhattanizing algorithms available, specified if *mode* is zero or nonzero.

When *mode* is zero (or not given), the operation works as follows. First, a polygon is decomposed into trapezoids, each of which is subdivided horizontally if necessary so that it can be further split vertically into rectangular and right-triangular pieces. The triangular pieces are divided, recursively, into a rectangular and two residual right-triangular pieces. All of the rectangular pieces whose height and width are *min_box_size* or larger are kept, and reassembled into a new Manhattan polygon.

In this mode, the rectangular elements can have arbitrary size, (though sufficiently large), and there is no restriction on coordinate locations.

When *mode* is nonzero, a different approach is taken. First, a polygon is decomposed into a collection of trapezoids, and each trapezoid is processed. For each trapezoid, all coordinates are moved to a "grid" of size *min_box_size*. If either side is non-Manhattan, Bresenham's method is used to scan the trapezoid vertically, creating a new Manhattan trapezoid for each "scan line" (grid point) where the width changes. The collection of trapezoids produced is reassembled into a new Manhattan polygon.

In this mode, all coordinates are moved to the grid, thus all the rectangular elements used to build the trapezoid have height and width an integer multiple of *min_box_size*.

19.13.17 The !polyfix Command: Fix Polygon

Syntax: `!polyfix`

This command will remove duplicate and in-line redundant vertices from selected polygons. In addition, it will repair the following conditions:

- If a reentrancy condition can be avoided by moving a vertex by one database unit, the vertex will be moved.
- If a "needle" vertex is found, it will be removed. A needle vertex is a vertex where the path doubles back on itself.

19.13.18 The !polyrev Command: Reverse Polygon Winding

Syntax: `!polyrev`

This will reverse the order of vertices of all selected polygons, i.e., changing the winding from clockwise to counter-clockwise and vice-versa. This should rarely if ever be needed.

19.13.19 The !noacute Command: Eliminate Acute Angles

Syntax: `!noacute`

This command will look at each currently selected polygon. For vertices that form an acute angle, vertices will be added so that no angle is acute, i.e. the sharp point is clipped off. This command is useful for preprocessing the database for flash conversion or other functions where acute angles are undesirable. It does not prevent DRC errors, and in fact may produce them. It also produces tiny (order of the layer's minimum dimension or one micron, if the minimum width for the layer is not given) changes to the layout. For example, consider a group of five or more polygons, each one of which is a pie section, that together form a disk. Running this command will produce a hole in the center, where the angles are clipped.

The algorithm works as follows. For each vertex V_n of a polygon, check the angle formed with adjacent vertices V_{n-1},V_{n+1}. If the angle is acute, construct a circle around V_n where the radius is the minimum of the layer's minimum dimension or the distance to the nearest of V_{n-1},V_{n+1}. Find the intersections of the circle with segments V_n,V_{n-1} and V_n,V_{n+1}. Replace the vertex V_n with these two points.

19.13.20 The !togrid Command: Move To Grid

Syntax: `!togrid`

This will move all vertices in selected boxes, polygons, and wires to the nearest snap point, using the grid/snap defined for the main window. There is no effect on subcells or labels. If the new object can not be created due to it having zero area, the old object is untouched. Duplicate vertices are removed from the new objects. Objects with vertices that are off-grid can change size and position due to this function.

19.13.21 The !tospot Command: Modify for Spot Size

Syntax: `!tospot` [*spotsize*]

When an e-beam mask is written, the layout is rendered using a certain pixel size (known as the "spot size") set by the e-beam equipment. This size may be as large as 0.5 microns but is typically much smaller, with smaller sizes providing higher resolution, but taking longer to write and therefor costing more. There can be numerical problems in "rasterizing" round objects to the e-beam grid. Since the round object is rendered as a collection of spot-pixels, the feature is not particularly round, but most importantly the number of pixels used may not be well defined, and therefor the figure area may not be as expected, or vary depending on position or rotation. *Xic*

The **!tospot** command will apply an algorithm (described below) to all selected polygons. The *spotsize*, if given, is the spot size to use in microns. Values up to 1.0 micron are accepted. If not given, the value is taken from the SpotSize variable if set, or the value of the `MfgGrid` from the technology file if SpotSize is not set.

The algorithm is intended to translate small objects with many vertices to a representation which will pass unchanged through e-beam rasterization. This will in general change the shape of an object, to something close to that which will be rendered on the mask.

The algorithm uses the following logic:

1. Find the bounding box of the figure.
2. Snap the box edges to the nearest spot boundaries.
3. If the center of the bounding box has changed, apply the same offset to the figure to keep it centered in the new bounding box.
4. Shrink the box by 1/2 of the spot size.
5. Clip the figure to the new bounding box.
6. For each vertex, move the vertex to the center of the closest spot.
7. Remove duplicate vertices.
8. Save the modified figure in the database.

Following application of the algorithm, each vertex of the figure is centered in an e-beam spot, so it is unlikely that round-off or other error will cause the figure to change during rasterization.

The algorithm is intended for unconnected, nonconducting objects such as vias. It should not in general be applied to wiring objects, since it will generate small gaps between processed objects which were originally touching, which will cause the extraction functions to detect that the objects are disconnected.

Although the object is shown on-screen as a polygon, The actual rendered object will be composed of pixels. The size of the object on-screen is therefor one spot-size smaller than the rendered size (since half of the spot for each edge is not shown).

Applying **!tospot** to circular objects created with a SpotSize is *not* the same as creating the circular object with the **round** or **donut** buttons with SpotSize nonzero. When using **!tospot** on round objects created without SpotSize set, it is best to use an even number of sides for round objects. In particular, an 8-sided figure is probably the best choice for a "circular" via.

19.13.22 The !origin Command: Move Cell Origin

Syntax: `!origin` *x y* `| n|s|e|w|nw|ne|sw|se`

In physical mode, this will move the cell origin. This applies a translation to every object in the cell, and rebuilds the database. The operation is more efficient than selecting everything and applying a move command, however there is no automatic "undo", except by applying the reverse operation.

All instances of the cell will change position if the cell origin is changed.

If the arguments are a coordinate *x,y* pair, the origin is shifted to that position (in microns) relative to the lower left corner of the cell's bounding box.

Alternatively, the argument can be one of the following compass directions:

`n`	The origin is moved to the top of the bounding box, the left/right position does not change.
`s`	The origin is moved to the bottom of the bounding box, the left/right position does not change.
`e`	The origin is moved to the right side of the bounding box, the up/down position does not change.
`w`	The origin is moved to the left side of the bounding box, the up/down position does not change.
`nw`	The origin is moved to the upper left corner of the bounding box.
`ne`	The origin is moved to the upper right corner of the bounding box.
`sw`	The origin is moved to the lower left corner of the bounding box.
`se`	The origin is moved to the lower right corner of the bounding box.

19.13.23 The !import Command: Import Cell Data

Syntax: `!import` *cellname*

In physical mode, this will move the contents of the physical part of *cellname* into the physical part of the current cell (the electrical parts are unchanged). The physical part of *cellname* will be empty after the operation. The coordinates of the objects are the same after the move, with respect to the origin of the current cell. This operation is not undoable.

19.14 Layout Information

19.14.1 The !fileinfo Command: Show File Statistics

Syntax: `!fileinfo` *filename* [*flags*] [*outfile*]

This will print information about the archive file given as the first argument. The output will go to a text file in the current directory.

The optional second argument is an integer or string which determines the type of information to print. If an integer, the bits are flags that control the possible data fields and printing modes. The string form is a space or comma-separated list of text tokens or hex integers. The hex numbers or equivalent values for the text tokens are or'ed together to form the flags integer. If the string contains white space it must be quoted.

The flag keywords and values are described with the `ChdInfo` script function in F.4.10.

If not given or given as 0, all flags except for `allcells`, `instances`, and `flags` are taken as set. This avoids printing the lengthly cells/instances list by default. The keyword `all` or value `-1` can be used to obtain all available information.

If the *outfile* is not given, the output will go to a file named "`xic_fileinfo.log`" in the current directory, otherwise it will go to the given file. In either case, the user is prompted to view the file when the operation is complete.

The operation has no effect on the database.

This command creates a Cell Hierarchy Digest (CHD) data structure for the given file, and uses the CHD to obtain the information in a very similar manner to the `ChdInfo` script function. In the **!fileinfo** command, the keyword flags listed below will show as indicated, as for the `FileInfo` script function:

`scale`
This will always be 1.0.

`alias`
No aliasing is applied.

`flags`
The flags will always be 0.

19.14.2 The !summary Command: Print Hierarchy Info

Syntax: `!summary` [`-v`] [*filename*]

This prints summary information (similar to the **Info** command) for each cell in the hierarchy rooted in the current cell to a file. If `-v` is given, the output will be more verbose. If no *filename* is given, a file named "`xic_summary.log`" will be created in the current directory.

19.14.3 The !compare Command: Compare Hierarchies

Syntax: `!compare` *arguments*

This function compares the geometry and instance placements in cells from two cell hierarchies, or between a cell hierarchy and cells in memory, or between cells in memory. It is also possible to compare properties of cells, cell instances, and objects. The results are written to a log file. It is used as a back-end for the **Compare Layouts** panel, and can be used directly.

There are three basic comparison modes. The per-cell object mode compares cell content object-to-object. A difference will be indicated if a given object does not have an exact counterpart in the other cell. The per-cell geometry mode does not look at objects, but rather considers the area occupied by the objects. Thus, differences will be indicated only if the covered area differs. The third comparison mode logically flattens the hierarchy before comparing the geometry. Thus, differences will be indicated only if the flat geometry (i.e., the mask layout) differs.

The results are written to a file named "`diff.log`" in the current directory. Each object or region that appears in one cell and not the other corresponding cell is written in a CIF-like format to the log file, unless the `-d` (diff only) option is given.

When the comparison finishes, the user is given the option to view the log file. The **!diffcells** command can be used to create cells from the log file for visualizing the differences.

Common Options

There is a large number of arguments that can be applied to set various modes and provide further input. These arguments must be given as separate tokens, and all start with a '`-`' symbol. The following options apply to all comparison modes.

`-f1` *source1*
: This is the "left" source. It is either the name of an archive file, or the access name of a Cell Hierarchy Digest (CHD) in memory, or a path to a CHD file. This argument is not mandatory, and if missing implies that cells listed for the left source are found in main memory.

`-f2` *source2*
: This is the "right" source. It is either the name of an archive file, or the access name of a Cell Hierarchy Digest (CHD) in memory, or a path to a CHD file. This argument is not mandatory, and if missing implies that cells listed for the right source are found in main memory.

For backward compatibility, the "`-f1`" and "`-f2`" are optional. If otherwise unassociated strings appear in the command line, the first will be taken as if given with `-f1`, the second (if any) will be taken as if given with `-f2`.

If a layout file name is given as a source, a temporary CHD will be created in memory and destroyed on command exit. Thus for repeated comparisons using the same file, it is more efficient to create the CHD first, and pass its name to this command.

`-c1` *cellname ...*
: This is a list of cell names found in the left source. If more than one name appears, the list should be quoted using double-quote marks. If no left source was given, the names should match cells in memory.

`-c2` *cellname ...*
: This is a list of equivalent cell names found in the right source. If more than one name appears, the list should be quoted using double-quote marks. If no right source was given, the names should match cells in memory.

The actual list of cells to compare is generated by logic to be described. The left source is taken as the "reference" for cell list creation.

In many cases, there is only one list of cells to compare (given in `-c1`), and each cell is sought in both sources. If a cell is found in one source and not the other, this will appear in the log file, but is not considered to be an error.

If a `-c2` "equivalence" list is given, there must be exactly the same number of entries as given in the `-c1` list. The cells in the two lists will be compared term-by-term, in order. This is how one can compare cells with differing names. In all other cases, the `-c2` list should not appear. It is an error if `-c2` is given without `-c1`, or the list lengths differ. However, the `-c2` list is ignored if in a per-cell comparison mode and the `-h` (recurse) option is given.

The interpretation of a non-existing `-c1` list depends on the comparison mode. If in flat comparison mode, or in a per-cell mode and the `-h` (recurse) option is given, then the effective cell list contains only the default cell from the left source. If this was a CHD name, the default cell is the one configured into the CHD, or the first top-level cell found in the source file. In the other cases, a missing `-c1` list is interpreted as all cells found in the left source.

In the special case that neither a left or right source is specified, then the `-c1` and `-c2` lists can not be empty, and the names are cells in memory to compare.

In the per-cell modes with `-h` (recurse) option given, each entry in the `-c1` list is hierarchically expanded to a full list of the cells under the given cell, and these names are merged into a new list that contains no duplicates. If no `-c1` list was given, per the discussion above, the cell list is effectively the hierarchy of the default cell from the left source. The recurse option can not be used unless a left source is specified, i.e., the left cells can't be from memory.

`-l` *layer_list*
: The *layer_list* is a space-separated list of layer names, which must be quoted if more than one layer appears. If no *layer_list* is given, all layers will be checked for differences.

`-s`
: If a *layer_list* is given, differences will be recorded in all layers **except** the layers in the *layer_list*.

`-d`
: Don't record the actual differences, only whether or not the cells differ. This only accounts for geometrical differences, properties are ignored.

`-r` *max_diffs*
: The integer *max_diffs* sets the maximum number of differences to allow before the comparison terminates. If not given or given a value 0, there is no limit. Beware that errors in the cell list could potentially lead to enormous output, so it is usually advisable to put a limit on the number of differences recorded.

The following options set the comparison mode. The per-cell comparison modes are generally faster and use less memory than the flat mode, since only the geometry from the two cells being compared is called into memory. The flat mode is required if the two layouts have differences in hierarchy.

`-g`
: When `-g` is given, per-cell geometric comparison is used. All "real" objects (boxes, polygons and wires) are considered when comparing geometry, text labels are ignored.

`-f`
: The `-f` option indicates flat comparison mode, and will supersede `-g` if also given. In flat comparison mode, geometry is logically flattened before comparison.

If neither `-f` or `-g` appears in the argument list, per-cell object mode is used.

Per-Cell Object Mode Options

`-t` *obj_types*
: The *obj_types* is a word containing any or all of the letters `c,b,p,w,l` which indicate cells, boxes,

polygons, wires, and labels. The letters indicate the types of objects that will be considered. If this option is not given, the default is "`cbpw`", i.e., labels are ignored.

Comparison of labels can lead to false differences when comparing cells read from different file formats, since label bounding boxes are not well defined across file format conversion.

`-b`
When given, a two-vertex wire or four-vertex polygon that is rendered as a Manhattan rectangle will match a rectangle object with the same dimensions. Thus, files that have had these features converted to boxes to save space can be directly compared, without a lot of spurious entries in output.

`-n`
When given, if duplicate objects are present in one or both of the files, unmatched duplicates will not be reported if one of the duplicates has a match. Thus files with duplicates removed can be compared with the original file, and the duplicates will not appear in output as differences.

`-x`
Expand subcell arrays (if comparing subcells). Cell arrays are converted to individual placements before comparison, avoiding false errors between arrayed and equivalent unarrayed layouts.

`-h`
The cell list is expanded so that all cells in the hierarchy under the given cells are compared. The left source is used to extract the hierarchy cells. The left source must have been specified, this option does no apply if the left cells are in memory.

`-e`
If `-e` is given, electrical cells will be compared. Otherwise, physical cells are compared.

Property comparisons are available only in per-cell object mode. Property lists of cells, instances, and objects can be filtered by property number and compared. Only the property lists of otherwise identical instances or objects will be compared. Property comparison is turned off by default, but can be enabled with the `-p` option.

`-p` *spec_word*
This option will set up property list comparison, which is available in per-cell object comparison mode. The *spec_word* is a collection of characters from the list below, order is unimportant.

`b, p, w, l, c, s`
The presence of these letters enables property list comparison between boxes, polygons, wires, labels, instances, and cells. The indicated object type or instance must also be enabled for checking with the `-t` option or by default, or the letter is ignored. The `s` character will always enable comparison of the property lists of the two source cells.

`n,u`
These two letters control the filtering applied to property lists before comparison. The filters limit the properties to compare. If `n` is given, no filtering is applied, so that all properties will be considered. This overrides `u` (below) if both are given.

If `u` is given, custom filtering will be applied. There are separate filters available for properties of cells, instances, and objects, for both physical and electrical comparisons. Custom filtering can be set up through the **Custom Property Filter Setup** panel, or by directly setting the corresponding variables. See the description of the panel in 14.13.3 for complete information.

If neither of these letters appear, default filtering is applied. For physical data, the default filtering action is no filtering. For electrical data, filtering is applied to cell and instance properties, and object properties are ignored, so that difference reporting applies to user-defined properties only.

Properties are compared by number and string. In the output file, property comparison result lines are all in comment form (with '`#`' as the first character) so that they will be ignored if the file is subsequently processed with the **!diffcells** command. Property comparison results consist of a string indicating the cell, instance, or object containing the properties. If an instance or object, this is common to both input sources. Following this are listings of properties found in one source and not the other. Properties that are identical in the two sources are not listed.

Per-Cell Geometry Mode Options

All of the options for per-cell object mode are available and have the same function, except that the only code that is considered for `-t` is "`c`". By default, subcell checking is not enabled. If enabled ("`-t c`" is given), then subcell placements are checked as in per-cell object mode.

When using per-cell geometry mode, the geometry is compared within areas of a grid whose size is given by the PartitionSize variable. Experimenting with this size can lead to improved speed, depending on the layout density. The default partition size is 100 microns. For best performance, this can be increased for low density, or reduced for high density, where "density" refers to the number of trapezoids per area.

Flat Mode Options

None of the per-cell options apply in flat mode, though with the exception of `-e` if given they will be benignly ignored. Flat mode applies only to physical data, and if `-e` is given, an error will result.

In flat mode, both *source* tokens must be provided, as flat comparison to memory cells is not available.

`-a` *L,B,R,T*
The `-a` option specifies the rectangular area where comparison is performed. If not given, comparison is performed over the entire cell area of both cells. The word that follows `-a` consists of the four rectangle coordinate values, in microns, separated by commas. There can be no white space.

The flat geometry mode is somewhat orthogonal to the other modes. The algorithm uses two levels of gridding to partition the layout into pieces, and directly compares the geometry in each fine grid cell. This is very similar to the algorithm described for the `ChdIterateOverRegion` script function.

`-i` *fine_grid*
This sets the size of the fine grid used for comparison. The geometry in each fine grid cell is compared. The value is in microns in the range 1.0 – 100.0, if not given 20.0 is used.

`-m` *coarse_mult*
This sets the size of the coarse grid, as an integer multiple of the fine grid size. The coarse grid size is the chunk size for reading geometry into memory. Once in memory, the geometry is split into the fine grid cells and compared. Using too large of a coarse grid can cause memory exhaustion for dense layouts, but on the other hand a larger coarse grid size usually improves speed. The user should experiment to find the best values for the fine and coarse grid for their layouts. The acceptable range for this parameter is 1 – 100. If not given, 20 is used.

19.14.4 The !diffcells Command: Create Cells from Comparisons

Syntax: `!diffcells` [*filename*]

This command will read a file produced by the **Compare Layouts** panel or the **!compare** command, and generate cells in the current symbol table containing the difference objects. If no *filename* is given, a file named "`diff.log`", in the current directory, will be read. Otherwise, the given file will be read, which should contain comparison output in the format of the `diff.log` file produced by the comparison commands.

The new cells are given the name of the source cell with a suffix "`_df12`" or "`_df21`". The "12" cells contain the objects found in the "`<<<`" cell but not the "`>>>`" cell, and vice-versa for the "21" cells. The created cells contain only geometry, so do not have subcells, and instance differences are ignored.

This can be very useful for graphically displaying the differences between cells.

19.14.5 The !empties Command: Check for Empty Cells

Syntax: `!empties` [`force_delete_all`]

This command will search through the hierarchy rooted in the current cell, and list the empty cells. Only the names of cells that have no content (objects or subcells) in either electrical or physical mode are listed. This test is performed automatically when a new cell is opened for editing/viewing, though this can be suppressed by setting the NoCheckEmpties variable.

Instances of empty cells are shown on-screen as a small highlighting box at the placement location. If empty cells are found, the **Empty Cells** pop-up appears, which provides a means for their deletion. The deletion capability is available in the *Xiv* feature set as well, in a rare instance where database changes are allowed. A list of the empty cells is shown, each followed by "yes" or "no", where "yes" implies that the cell will be deleted. Initially, all listings will be "no", but these can be changed by clicking on them. The **Delete All** button sets all entries to "yes", and the **Skip All** button sets all entries to "no". Pressing **Apply** will actually perform the deletions.

However, is is not possible to delete instances of empty cells that are contained in a parent cell with the IMMUTABLE flag set. Cells referenced by an instance in an immutable parent will not be deleted, however instances in non-immutable parents within the hierarchy will be deleted.

If cells are deleted, the search for empty cells is repeated, and the pop-up will be updated if any are found. Additional cells may become empty due to the previous deletions.

If the literal "`force_delete_all`" argument is given, all empty cells in the hierarchy, including those that become empty due to prior deletions, will be deleted (if possible). The pop-up will not appear.

The current cell, if empty or if it becomes empty, will not be deleted.

19.14.6 The !area Command: Measure Layer Area

Syntax: `!area` [*layername*]

The **!area** command prints the area (in square microns) covered by the given layer, in the current cell and all of its descendent cells. If *layername* is not given, the current layer is used, if in physical mode.

Only physical mode layers can be given, and only physical cells are computed. This does *not* account for overlapping objects.

19.14.7 The !perim Command: Measure Object Perimeter

Syntax: `!perim`

This command will compute the perimeter of selected objects and subcells and print the totals, in microns. Labels are ignored. Separate totals are given for subcell perimeter, and for the perimeter of geometric objects.

19.14.8 The !bb Command: Print Bounding Box

Syntax: `!bb`

In physical mode, this prints the bounding box coordinates of the current cell, in microns.

19.14.9 The !checkgrid Command: Mark Off-Grid Vertices

Syntax: `!checkgrid` [`c`] [`o`] or
`!checkgrid` [`-`] [`-l` *layer_list*] [`-s`] [`-g` *spacing*] [`-b` *L,B,R,T*] [`-t` *bpw_string*] [`-d` *depth*] [`-f` *outfile*]

This is really two commands in one. The first mode checks objects in the current cell, and will mark off-grid vertices on-screen. The second mode will check vertices to all levels of the hierarchy.

The first form will mark vertices of objects and cells that are off-grid. The reference grid is the grid currently applied in the main drawing window. If there are selected objects, these (only) will be tested. Objects or subcells that have an off-grid vertex will remain selected, other objects will be deselected. If no testable objects are selected, all objects on visible, selectable layers will be tested. Cells will be checked if the '`c`' modifier is given. Objects or cells that have an off-grid vertex will be selected, and all off-grid vertices will be marked.

Giving the **!checkgrid** command with the '`o`' modifier (or '`n`' or '`0`' (zero)) will remove the marks from the screen.

If the first character of the argument string is '`-`', the second mode will be used. An argument containing a single '`-`' is valid to enforce this. The other possible arguments are listed below. All of these are optional.

The command will look at objects in the hierarchy, and if an object vertex would appear off-grid in the current cell, it will be listed in an output file.

`-l` *layer_list*
: The argument is a space-separated list of layer names, which should be quoted if it contains more than one entry. Only objects on the listed layers will be checked, or if `-s` is also given objects on layers not listed will be checked. If not given, all layers will be used.

`-s`
: If a *layer_list* was given, objects on these layers will be ignored.

`-g` *spacing*
The *spacing*, in microns, is the assumed grid spacing. If not given, the value from the current grid setting will be used.

`-b` *L,B,R,T*
This specifies a rectangular region in the current cell where testable objects will be searched for. If not given, the entire cell will be searched. The coordinates are in microns, separated by commas with no white space.

`-t` *bpw_string*
This is a string consisting of one or more of the letters "`b`", "`p`", and "`w`". This indicates the type of objects to test: boxes, polygons, and wires. If not given, "`bpw`" is assumed.

Note: only the lower left and upper right vertices of boxes are tested, since the other two are redundant.

`-d` *depth*
This sets the maximum hierarchy depth to search for objects. If not given, all levels of the hierarchy will be searched. A zero value would search only the current cell.

`-f` *outfile*
This sets the name of the output file, which will contain a sorted list of off-grid vertices. If not given, the name of the current cell, suffixed with "`_vertices.log`", will be used. If the name is "`stdout`", output will go to the standard output (console window).

19.14.10 The !checkover Command: Report Subcell Overlap

Syntax: `!checkover [`*filename*`]`

This command creates a report of subcell overlap in the current physical cell. The report is written to the given *filename*, or to a temporary file if no name is given. The user is given the option to view the report, if a filename is given, otherwise the file viewer pops up automatically for the temporary file, and the temporary file is deleted.

19.14.11 The !check45 Command: Select Non-45 Polys and/or Wires

Syntax: `!check45` [`p|w`]

This will select polygons and/or wires in the current cell that have an angle that is not an exact multiple of 45 degrees. If an argument "`p`" is given, only polygons are checked, or if the argument is "`w`" only wires are checked. Otherwise both polygons and wires are checked. Only objects on visible, selectable layers are checked.

19.14.12 The !dups Command: Select Coincident Objects

Syntax: `!dups`

This checks the current cell for identical objects placed on top of one another. The duplicate objects are selected. This command initially deselects anything previously selected.

19.14.13 The !wirecheck Command: Check Wires

Syntax: `!wirecheck` [*layer ...*]

Wire database objects have the property that their geometric shape is not unambiguously specified. Every tool contains code that generates a polygon from the wire vertex list, which can be displayed and further processed. The details of how corners are handled, and how the "rounded" end style is handled, can vary slightly between tools.

Some wires are difficult to represent as a polygon, and in fact may cause failure with some tools (and possibly not others). Although wires sensibly created by hand would rarely if ever cause trouble, wires generated by format converters or some other program might cause failures, for example when "fracturing" the layout file during mask generation. Even wires that look reasonable on-screen may not be renderable on other tools, thus *Xic* provides some tests that can be applied to flag potential problems.

Wires can be "questionable" or "bad". Bad wires can not be rendered, and will never be included in the *Xic* database. These wires are always flagged as errors when seen.

Wires that are "questionable" have vertices that are closely spaced compared to the wire width, and trigger an edge-clipping fix-up in the wire-to-polygon function. Such wires may cause rendering difficulty in other tools. In addition, wires whose polygon representation requires more than 600 vertices are flagged as questionable.

When reading a layout file, questionable wires will be reported as warnings in the log file.

This command can be used to find questionable wires in the current cell. It takes a list of layer names as arguments, which will limit the testing to wires on those layers. If no arguments are given, all layers will be used.

If wires are selected before the command is given, only the selected wires on the given layers (or on any layer, if no arguments are given) will be checked. If no wires are selected, all wires on the layers given (or on any layer if no arguments are given) will be checked.

If a wire is determined to be questionable, it will be, or remain, selected. The **Info** command in the **View Menu** can be used to determine the exact nature of the defect.

The flags that might be listed in the info for wires have the following explanations.

`ONEVERT`
: The wire consists of a single vertex only. The interpretation of this case may be tool-dependent.

`ZEROWIDTH`
: The wire has zero width. Zero width wires have no physical significance and should not appear in a physical layout, though generally they are simply ignored.

`CLOSEVERTS`
: The wire contains at least two vertices whose spacing is less than half of the wire width. This may not be a problem, however wires that are difficult to render will always have this condition.

`CLIPFIX`
: This flag indicates that special fix-up code was triggered when the representing polygon was created, which indicates that rendering requires non-trivial processing. Wires that have this flag are suspect (they will also always have `CLOSEVERTS` set).

`BIGPOLY`
: This flag indicates that the representing polygon contains more than 600 vertices. This is not really a problem, by does indicate that the wire may be overly complex.

Wires that are determined to be questionable will have one or more of `ZEROWIDTH`, `CLIPFIX`, or `BIGPOLY` set.

19.14.14 The !polycheck Command: Check Polygons

Syntax: `!polycheck` [*layer ...*]

This command will test polygons for reentrancy and other defects. It takes a list of layer names as arguments, which will limit the testing to polygons on those layers. If no arguments are given, all layers will be used.

If polygons are selected before the command is given, only the selected polygons on the given layers (or on any layer, if no arguments are given) will be checked. If no polygons are selected, all polygons on the layers given (or on any layer if no arguments are given) will be checked.

If a polygon fails the test it will be, or remain, selected. The **Info** command in the **View Menu** can be used to determine the exact nature of the failure.

Duplicate vertices will be silently removed from the checked polygons.

The polygons may be repairable with the **!polyfix** command.

19.14.15 The !polymanh Command: Select Manhattan Polygons

Syntax: `!polymanh` [*arg*]

Without an argument, this command will deselect all polygons, and then select only those that are Manhattan. If there is an argument, which can be any text token, the non-Manhattan polygons will be selected instead.

19.14.16 The !poly45 Command: Select Non-45 Polygons

Syntax: `!poly45`

This will select polygons in the current cell that have an angle that is not an exact multiple of 45 degrees. All polygons on visible, selectable layers are checked.

This is equivalent to using a "`p`" argument with the **!check45** command.

19.14.17 The !polynum Command: Number Vertices

Syntax: `!polynum` [*arg*]

This function activates a mode where the vertex numbers of selected polygons are shown on-screen. If no argument is given, the display mode is toggled. If the argument is "`y`", "`1`", "`on`", etc., the display mode is enabled. If the argument is "`n`", "`0`", "`off`", etc., the display mode is disabled.

19.14.18 The !setflag Command: Set Internal Cell Flags

Syntax: `!setflag` *name* `0|1`
Syntax: `!setflag ?`

This allows the flags associated with the current cell to be changed. The second form of the command brings up a window containing a list of the flag names and descriptions, as does **!setflag** without arguments.

The IMMUTABLE and LIBRARY flags can also be modified with the **Set Cell Flags** pop-up from the **Cells Listing** panel.

The IMMUTABLE flag will also control availability of user interface features associated with cell editing. This flag is also set by the **Enable Editing** button in the **Edit Menu**.

19.15 Libraries and Databases

19.15.1 The !mklib Command: Create Library File

Syntax: `!mklib [`*archive_file*`] [-d] [-a] [-l]|[-u]`

This command will create or append to a library file adding references to cells in the current hierarchy, or to cells in an archive file if *archive_file* is given. If `-a` is given, the library entries will be appended to an existing library, otherwise a new library will be created. If `-d` is given, a `Directory` reference will be created. These are usually collections of native *Xic* cells. The *archive_file* argument is the directory path in this case, if given. If not given, the path will be prompted for. The `-l` and `-u` arguments are ignored with `-d` given. Otherwise, if `-l` is given, the reference name will be a lower-cased version of the cell name, or, if `-u` is given, the reference will be upper-cased.

The following applies when `-d` is not given.

if *archive_file* is given, all cells found in the file will be added to the output library as references. If the file is not rooted, a reference directory is prompted for. This is the full path to the directory containing the archive file. The prompt is skipped if *archive_file* is rooted.

If *archive_file* is not given, and the current cell was read from an archive file, the user is prompted for the name of a reference archive file. If a name is given, the library entries will be in the form

`Reference` *refname reference_path/name cellname*

otherwise the references are in the form

`Reference` *refname reference_path/cellname*

as for native cells. The user is next prompted for the reference path. This should be the path to the directory where the referenced cell files, or archive file, reside. The current directory is the default. Finally, the user is prompted for the name of the library file, which is then created, or appended to if it exists and `-a` was given.

Example

You have a GDSII file named `/usr/local/cad/standard_cells/std_cell_lib.gds` and you want to enable the standard cell definitions in *Xic* as library cells. This is a two step process.

1. First create a library file with the command

   `!mklib /usr/local/cad/standard_cells/std_cell_lib.gds`

   This creates a file named `std_cell_lib.lib` in the current directory. Move this file to a directory in your cell search path if desired (the current directory is probably in the search path). You may want a separate directory for library files, for example.

2. The library will need to be opened in order for cells in the library to resolve references as designs are read. From the **Libraries** panel from the **Libraries List** button in the **File Menu**, double click the folder icon for the `std_cell_lib.lib` entry. The icon will change to an open folder, indicating that the library is now open.

   You can add an `OpenLibrary` call to your `.xicstart` file, to open the library automatically whenever *Xic* starts. Otherwise, you will need to open it manually when needed.

19.15.2 The !lsdb Command: List Special Databases

Syntax: `!lsdb`

This command pops up a list of the "special" databases currently in memory, by name and type. These are the databases created by the `ChdOpenOdb`, `ChdOpenZdb`, and `ChdOpenZbdb` script functions. Special databases are also used internally, for example in the **Cross Section** command from the **View Menu**.

19.16 Marks

19.16.1 The !mark Command: Create User Marks

Syntax: `!mark l|b|t|u|c|e|d|w|r [`*attr_flags*`]`

This command allows the user to add annotation marks to the cell display, physical or electrical. These marks are not part of the design and will not be saved in output, but are useful for temporarily marking or highlighting an area for reference. They will appear on plots of the cell.

The marks are persistent to a cell, meaning that they will appear whenever the cell is displayed as the top-level cell in a window. Each cell in memory can have its own set of marks. Marks are not displayed in expanded subcells.

The first argument is a letter giving the initial type of mark to create. When the command is active, any of these letters may be typed in a drawing window, which will change the current mark type. While the command is active, clicking twice or dragging will produce a mark, and Shift-clicking in an existing mark will delete the mark.

The optional *attr_flags* is a decimal number representing flags bits that control presentation format of the mark. The bits are

bit 0
: When set, a dashed line is used, otherwise solid.

bit 1
When set, the mark will blink.

bit 2
When set, an alternate color will be used for the mark (bit 1 is ignored). The default is the normal highlighting color.

The value is a digit representing the set bits, for example 3 sets bits 0 and 1, 5 sets bits 0 and 2, etc. A value 0 is the default.

When the command is active, pressing a digit key will reset the current attribute flags for subsequent marks.

The following marks are available:

`l`
Draw a line. Click twice or drag to define the line endpoints.

`b`
Draw an open box. Click twice or drag to define the box boundary.

`t`
Draw an open "horizontal" triangle, with the base a vertical line, and the third point pointing to the left or right at the midpoint of the base. The triangle will fit inside of the ghost-drawn box shown during creation. The initial press location sets the x coordinate of the triangle base.

`u`
Draw an open "vertical" triangle, with the base a horizontal line, and the third point pointing up or down at the midpoint of the base. The triangle will fit inside of the ghost-drawn box shown during creation. The initial press location sets the y coordinate of the triangle base.

`c`
Draw an open circle. The press location is the center of the circle, and the distance to the second point sets the radius.

`e`
Draw an open ellipse. The ellipse will fit inside of the ghost-drawn box shown during creation.

To delete a mark, while the **!mark** command is active, click on the mark to delete with the **Shift** key held. Any mark under the click location will be deleted, not just those of the current type.

Marks can be saved to a file, and restored from a file. This is accomplished by giving the following code letters, which can appear in the same contexts as the mark code letters.

`d` or `w`
The user will be prompted for the name of a file, then the existing marks in the current cell will be written to the file.

`r`
The user will be prompted for the name of a file, which should be in the format produced with the `d` or `w` option. If the file was produced for the same cell name and display mode of the current cell, the marks will be read from the file and added to the current cell.

The file format is not currently documented, but is very simple and should be easy to figure out by inspection.

The marks manipulated with the **mark** command are the same as the marks produced with the `AddMark` script function. Note that `AddMark` can create additional mark types not (yet) supported by the command interface.

19.17 Memory Management

19.17.1 The !clearall Command: Clear All Memory

Syntax: `!clearall`

This command will clear all program memory, no questions asked, similar to the `ClearAll` script function. Be careful, since anything cleared and not saved is gone forever. There is no current cell when the operation completes, so that a new cell must be opened explicitly.

19.17.2 The !vmem Command: Windows Virtual Memory Info

Syntax: `!vmem`

This command is available in Microsoft Windows releases only. It will print system virtual memory information in a pop-up window. This probably has very limited value to the user.

19.17.3 The !mmstats Command: Show Memory Manager Statistics

Syntax: `!mmstats`

The command will print, on the console window, statistics from the first-level memory manager. The first column is the internal name of a data structure being managed. The second column is the size of the structure in bytes. The remaining columns are:

`fl`	length of the full block list, each block contains 64 entries
`fh`	hash table width for full list entries
`nfl`	length of the not-full block list
`nfh`	hash table width for not-full list entries
`u`	number of bytes in use
`nu`	number of bytes allocated but not in use

This information is probably not of much value to the user.

19.17.4 The !mmclear Command: Clear Recycle Free Lists

Syntax: `!mmclear`

This will free the caches associated with the memory manager. Each managed data type has a cache of deleted objects, which are used to quickly service an allocation request. This command will clear the caches, giving the object memory back to the system. This is implicitly called by **!clearall** and the `ClearAll` script function.

19.18 OpenAccess Interface

19.18.1 The !oaversion Command: Print OpenAccess Release Number

Syntax: `!oaversion`

This command exists only when the OpenAccess plug-in is loaded.

This command will print, on the prompt line, the OpenAccess release number.

19.18.2 The !oadebug Command: Enable Logging

Syntax: `!oadebug` [`+|-`] [`l`[`oad`]] [`p`[`cell`]] [`n`[`et`]]

This function enables or disables logging of OpenAccess interactions and operations. There are three categories of messages.

`load`
Messages emitted when reading cell data from OpenAccess and building equivalent cell structures in *Xic*.

`pcell`
Messages emitted when instantiating parameterized cells.

`net`
Messages emitted when evaluating connectivity.

Each category can be separately enabled or disabled, depending on whether the keyword follows a '+" or '−'. An initial virtual '+' is assumed. Only the first character of the keyword needs to be given, and keyword recognition is case-insensitive. All keywords are initially disabled (no logging).

Example: turn on `net`, turn off `load`.

```
!oadebug n -l
```

With no arguments given, the command will print the present flag status on the prompt line.

The debugging output will go to a log file named "`oa_debug.log`" which will be located in the log files area. The **Log Files** button in the **Help Menu** will enable access to the log files.

The **Logging** button in the **Help Menu** brings up a panel from which the three OpenAccess logging flags can be set, as an alternative to using the **!oadebug** command.

19.18.3 The !oanewlib Command: Create New OpenAccess Library

Syntax: `!oanewlib` *libname* [*techlibname*]

This command exists only when the OpenAccess plug-in is loaded.

This will create a new library *libname* if it does not already exist. The *techlibname* is the name of an existing library, if given. The new library will attach to the same technology database as *techlibname*, or will attach to the local technology database found in *techlibname* if *techlibname* has no attachment. If *techlibname* is given then it must exist.

If *techlibname* is not given, then the technology will be attached from the library named in the OaDefTechLibrary variable, if that variable is set. If no technology source is found, the library will be created with an empty technology database.

If the library is created, it will be given a property which allows *Xic* to write into it. Setting or clearing of this property, or "branding" the library, can be controlled subsequently with the **!oabrand** command.

19.18.4 The !oabrand Command: Permit Save from *Xic* in OA Lib

Syntax: `!oabrand` [*libname* [`y|n`]]

This command exists only when the OpenAccess plug-in is loaded.

By default, OpenAccess libraries that were not created by *Xic* are read-only within *Xic*. This is due to the fact that overwriting Virtuoso views will destroy them for use with Virtuoso, and the same probably applies to files for other tools as well.

If the second argument is not given, the branded status of the named library is reported on the prompt line. Otherwise, this function will apply or remove the brand to a library. The second argument, if affirmative, will cause the brand to be applied. Otherwise, an existing brand will be removed. This argument can be any commonly known name for affirmation such as "`y`", "`yes`", "`true`", "`1`", etc. If not recognized as affirmative, it is taken as non-affirmative.

If no library is given, the variable OaDefLibrary is checked for a library name, which is used if set.

Libraries that are created by *Xic* are already branded. If needed, this command can be used to remove write permission for *Xic* by un-branding. This can also be used to brand a library created by another tool, allowing *Xic* to write into that library. The user must understand the risks involved.

19.18.5 The !oatech Command: Query OA Technology Database

Syntax: `!oatech` *cmd libname* [*args*]

This command exists only when the OpenAccess plug-in is loaded.

This function has a number of forms, corresponding to various actions to perform on the technology database. These forms, and the corresponding actions, are described below. In each case, the first character of the first token indicates the command type. If this is preceded by a hyphen, the hyphen will be ignored. Thus, for example, first arguments "`-p`", "`p`", and "`print`" are all equivalent.

[-]a[`ttach`] *libname fromlib*
: Attach the technology database from *fromlib* to *libname*. This will fail if *libname* has a local technology database, such must be destroyed first. If *fromlib* has an attachment, then *libname* will receive the same attachment, otherwise *libname* will attach to the local technology database in *fromlib*.

[-]d[estroy] *libname*

If *libname* has an attached technology database, the attachment will be removed. Otherwise, the local technology database will be destroyed.

[-]h[as_attached] *libname*

This produces a message on the prompt line indicating whether or not *libname* has an attached technology database, and if so, provides the name of the library supplying the technology database.

[-]p[rint] *libname* [-o *filename*] [*which* [*prname*]]

If *which* and *prname* are not given, a file in the format of a Virtuoso ASCII technology file will be produced, containing all technology information known to *Xic* from the technology database associated with *libname*.

If *which* is given (including "all"), the file format is not specific and a complete data dump of relevant data. This is intended for debugging and information searching.

Output goes to the console window by default, but the -o option, if given, signals that the following argument is a file name for output.

The *which* is a code indicating what type of information to print, and *prname* is a sub-type which applies to particular values of *which*. The prefixes understood for *which* are listed below, characters that follow the prefix are ignored. Recognition is case-insensitive.

prefix	**prname**	**will print**
"all"		everything
"u"		units
"an"	*	analysis libraries
"l"	*	layers
"o"	*	operating points
"p"	*	purposes
"si"	*	site definitions
"va"		values
"viad"	*	via definitions
"vias"		via specifications
"viav"		via variables
"co"	*	constraint groups
"cg"	*	constraint groups (as above)
"cp"		constraint parameters
"d"		derived layer parameters
"ap"		application object definitions
"g"	*	groups

For the types marked with an asterisk above, the *prname* is recognized as the name associated with the records of that type, and only the record with matching name, if any, will be printed. If *prname* is not given, all records of the selected type are printed.

[-]u[nattach] *libname*

If *libname* has an attached technology database, remove the reference.

19.18.6 The !oasave Command: Save Cell to OA Library

Syntax: !oasave [-a] [*libname*]

This command exists only when the OpenAccess plug-in is loaded.

Save the current cell to *libname*. If no library name is given, the variable OaDefLibrary is checked for a library name, which is used if set. If the `-a` option is given, the cell hierarchy under the current cell is written, otherwise only the current cell is written. The value of the OaUseOnly variable will limit the data written to electrical or physical. This tracks the setting ot the **Data to use from OA** radio button group in the **OpenAccess Libraries** panel.

19.18.7 The !oaload Command: Read Cell from OA Library

Syntax: `!oaload` [*libname* [*cellname*]]

This command exists only when the OpenAccess plug-in is loaded.

This will load the given cell and its hierarchy into *Xic*. If the *cellname* is not given, all cells found in the library will be loaded into *Xic*. If no library is given, the variable OaDefLibrary is checked for a library name, which is used if set. The value of the OaUseOnly variable will limit the data read to electrical or physical. This tracks the setting ot the **Data to user from OA** radio button group in the **OpenAccess Libraries** panel.

19.18.8 The !oadelete Command: Delete OpenAccess Object

Syntax: `!oadelete` *libname cellname* [*viewname*]

This command exists only when the OpenAccess plug-in is loaded.

The *viewname*, if given, can be an actual OpenAccess view name, or "`electrical`", or "`physical`". The latter two map into corresponding OpenAccess view names.

The indicated cell and view in the library will be destroyed. If the *viewname* is not given, all views for the cell will be destroyed.

Be careful, this operation can not be undone.

19.19 Parameterized Cells

19.19.1 The !rmpcprops Command: Remove PCell Properties

Syntax: `!rmpcprops` [`-a`]

Warning: this operation is not undoable.

This command applies to all cells in the hierarchy of the current physical cell. There are two passes made through the hierarchy. On the first pass, cells that are parameterized cell (pcell) sub-masters may have their pc_name and pc_params properties removed. This will be true for "foreign" pcells created in and imported from another tool or library such as OpenAccess, and if `-a` is given, this will also apply to native pcells. Once these properties are removed from a pcell sub-master. the cell becomes in all respects an ordinary cell.

On the second pass, the masters of cell instances that have pcell properties are checked, and if the master does not have pcell properties (they were likely removed in the first pass), the instance pcell properties are removed.

Running this command will remove any ambiguity about whether sub-master cells will be saved to an archive (they will always be saved, since they are now normal cells), and there will never be an attempt to resolve placements of the cells by executing a super-master (instances are no longer seen as pcell placements). All history that the cell was once created from a pcell super-master is gone.

This command is **not undoable**. Once the properties are stripped, there is no way to put them back, except perhaps very laboriously by hand. Don't use this command unless you want all pcell history in the current cell hierarchy to go away forever.

When importing design data from Cadence Virtuoso, for example, using the Express PCell feature to obtain pcell sub-masters, you may wish to use this command on the new hierarchy. In *Xic*, the pcells can not be evaluated anyway, and their presence may cause trouble. For example, if the hierarchy is saved to disk as a GDSII or other archive file, by default the sub-masters are **not** written. When reading this file at some future time, unless the Virtuoso database is present and able to provide the sub-masters, the pcell instances won't be resolved. Thus you must remember to explicitly enable saving the sub-masters when writing output, unless you have used the **!rmpcprops** command.

19.19.2 The !preload Command: Pre-Load PCell Sub-Masters

Syntax: `!preload` *archive*

Suppose that one has a collection of pcell sub-master *Xic* cells that have been imported from a foreign OpenAccess tool such as Virtuoso. These are assumed to not be portable pcells. One would like to use these cells to resolve pcells when reading directly from the OpenAccess database. There are two issues: 1) the system needs to know that these cells are available, and 2) one has to remap the cell names. The first issue is fixed simply by making the sub-masters available through the library mechanism. The second issue is due to the simple naming convention of the sub-master instantiations, which suffixes the pcell name with "`$$`" followed by an integer. The integer is a count of when the cell was generated, and is consistent with the design output at the time, but there is no guarantee the the names are consistent with the design at other times.

This command will read a collection of cells into a temporary symbol table. Those that are pcell sub-masters have the property strings entered into the internal pcell database, under the existing cell name. This will cause the correct cell name to be associated with a given parameter set. The cells are not saved, but the entries in the pcell table persist so that resolution, when reading OpenAccess or otherwise, will reference the correct cells. The cell collection must be available through an open library, and this function must be run before loading the design.

The argument is either a path to a directory containing native pcell sub-master cells, or a path to an archive file that contains the cells. This capability is also available with the `RegisterSubMasters` script function.

19.20 Rulers

19.20.1 The !dr Command: Delete Rulers

Syntax: `!dr` [*arg*]

This will delete currently displayed rulers, as generated by the **Rulers** command in the **View Menu**. If no *arg* is given, the most recently generated ruler is deleted. The *arg* can be an integer, or '`a`'. if '`a`' is given, all rulers for the current cell are deleted. If a number is given, that ruler, counting backward from the most recently generated, will be deleted, i.e., 0 erases the most recent ruler, 1 erases the one before that, etc.

19.21 Scripts

19.21.1 The !script Command: Add Script

Syntax: `!script` *name* [*path*]

This command will add *name* to the list of user-defined function buttons in the **User Menu**. When the button is pressed, the file indicated by *path* will be executed as script text. The *name* variable should be the actual name to appear in the menu. The *path* should be a full path to a file, which can be any file name as long as it contains a script, i.e., the `.scr` extension is optional. A script added that has the same name as a script in the technology file or the script path will supersede the previous script definition.

If no *path* is given, any command previously added with the **!script** command with the same name is deleted from the **User Menu**. This does not affect scripts defined in the technology file or in the script path, except that these are reverted to if their names matched an input to the **!script** command.

19.21.2 The !rehash Command: Rebuild User Menu

Syntax: `!rehash`

This command re-reads the script files and libraries along the script search path, and rebuilds the **User Menu**, the same as the **Rehash** button in the **User Menu**.

19.21.3 The !exec Command: Execute a Script

Syntax: `!exec` *script*

This command will execute a script. The argument is a string giving the script name or path. If the script is a file, it must have a "`.scr`" extension. The "`.scr`"" extension is optional in the argument. If no path is given, the script will be opened from the search path or from the internal list of scripts read from the technology file or added with the **!script** command. If a path is given, that file will be executed, if found. It is also possible to reference a script which appears in a sub-menu of the **User Menu** by giving a modified path of the form "`@@/`*libname*`/.../`*scriptname*". The *libname* is the name of the script menu, the ... indicates more script menus if the menu is more than one deep, and the last component is the name of the script.

19.21.4 The !lisp Command: Execute Lisp Script

Syntax: `!lisp` *filename* [*args* ...]

This is an interface to the Lisp/Skill parser that is under development. The *filename* is searched for in the script path and the current directory, and is expected to contain a script in Lisp format. The file will be parsed and the code executed.

Any text following the filename will be parsed as Lisp and included in the argument list. The argument list can be accessed from within the script through the global variables `argc` and `argv`.

`argc`
An integer giving the length of `argv`.

`argv`
A list. The first element is the file name, followed by the arguments if any.

See 5.7.1 for a description of the language implementation in *Xic*.

19.21.5 The !py Command: Execute Python Script

Syntax: `!py` *scriptfile args ...*

This command is available only if the Python interpreter plug-in has been loaded, and is not available under Microsoft Windows.

The arguments will be passed to the Python interpreter for evaluation.

19.21.6 The !tcl Command: Execute Tcl Script

Syntax: `!tcl` *scriptfile args ...*

This command is available only if the Tcl/Tk or Tcl-only plug-in has been loaded, and is not available under Microsoft Windows.

The command will execute a Tcl script (see 2.13), contained in the file given as an argument. Tk functions are not supported. Command arguments can be referenced in the script using the standard `argc`, `argv` mechanism. The language syntax is provided in documentation supplied with Tcl, and is described in several books. Much information can be found on the internet.

The *scriptfile* is expected to contain Tcl commands, and is linearly parsed and executed.

19.21.7 The !tk Command: Execute Tcl/Tk Script

Syntax: `!tk` *scriptfile args ...*

This command is available only if the Tcl/Tk plug-in has been loaded, and is not available under Microsoft Windows.

This command will execute a Tcl/Tk script (see 2.13), contained in the file given as an argument. Command arguments can be referenced in the script using the standard `argc`, `argv` mechanism. The language syntax is provided in documentation supplied with Tcl/Tk, and is described in several books. Much information can be found on the internet.

The *scriptfile* must have a `.tcl` or `.tk` extension, appropriate for the file contents. The Tk language is a superset of Tcl, containing a graphical interface. The files are executed differently: Tk files are executed in an event loop and a default window will be created, and execution will continue until all created windows are destroyed. Tcl files are interpreted linearly, with no graphics.

An example Tk script named "`tkdemo.tk`" is provided with the examples and can be used to set up and test the Tk execution facility.

19.21.8 The !listfuncs Command: List Saved Functions

Syntax: `!listfuncs`

This command pops up a list of the script functions that are currently saved in memory. All functions that *Xic* sees are saved.

19.21.9 The !rmfunc Command: Remove Saved Function

Syntax: `!rmfunc` *func_name_reg_exp*

This command allows functions to be removed from memory. The argument is a regular expression that should match one or more function names. Saved functions can be listed with **!listfuncs**.

19.21.10 The !mkscript Command: Create Current Cell Script

Syntax: `!mkscript` [`-d` *depth*] [*filename*]

This command writes a script file that will create the contents of the current cell, and its hierarchy to arbitrary depth. When executed, the script will create the cells, and place objects and subcells as needed to recreate the original cells.

This could be useful as a starting point for creating parameterized cells. It might also be useful to new users for learning the scripting language.

The function presently works in one mode only, i.e., you can generate a script that will build electrical or physical cells or hierarchies, but not both modes together. One could generate a script for each mode and combine them by hand, however.

The *depth* argument is an integer depth, with 0 being the default, which indicates to write the current cell only. The value -1 or a word starting with 'a' indicate all levels.

If a *filename* is given, output goes to that file. Otherwise, the script is written to "`mkscript.scr`" in the current directory.

Although things seem pretty solid for physical mode, electrical mode is far more complex and should be considered experimental at this point. There are probably things that don't work, for example mutual inductors probably won't be created. The situation should improve in time, though it is not clear if this feature is of much use in electrical mode.

Incorporation of this feature led to some significant updates in script functions and elsewhere for efficient support.

19.21.11 The !ldshared Command: Load Plug-In Script Library

Syntax: `!ldshared` *library* [*args...*]

This will load a script library plug-in as created with the `scrkit` provided with *Xic* distributions. The `scrkit` directory contains files and instruction for creating libraries of C/C++ functions which can be called from scripts.

The required argument is a path to the shared library file as generated from the `scrkit` system. Anything else in the line is passed to the library `init` function verbatim. The library author can add a parser for this, for providing initialization options if needed.

Loaded libraries can not be unloaded, but can be reloaded, perhaps after modification and recompile. If a library is reloaded, a **!rehash** is done, to eliminate bad function pointers to the library functions, which would likely crash the program if referenced.

19.22 Selections

19.22.1 The !select Command: Select Objects

Syntax: `!select` *what qualifier_or_regex* [*keyword expression*]

This command allows objects to be selected according to the specification provided. There is also a companion **!desel** command which deselects selected objects.

The values (literal) for *what* are:

`c`[ell]
`l`[ayer]
`n`[ame]
`m`[odel]
`v`[alue]
`p`[aram] or `i`[nitc]
`o`[ther]
`y`[...] (indicates nophys)

Only the first character of the token is significant. If '`c`' is given, the intended targets for selection are subcells. If '`l`' is given, the targets are objects on a specified layer. The remaining options specify electrical properties, which allows selection of devices with these properties. The `param` property was known in earlier releases as the `initc` (initial condition) property, both names are accepted.

The *qualifier_or_regex* is a pattern matching regular expression. This is expected to match the layer or cell name or property value as per *what.* All objects with a successful pattern match are selected. The layer qualifier consists of the layer regular expression, followed by the optional tokens

`b`[oxes]
`w`[ires]
`p`[olygons]
`l`[abels]

These specify types of objects that will be selected. For selecting objects on physical layers, an additional *keyword expression* pair can be included in the command. The complete syntax in this case is

> `!select l[ayer]` *layer_re* `[b[ox]] [w[ire]] [p[olygon]] [l[abel]] [`*keyword expression*`]`

For layers, the hyphen ('-') is an alias for the current layer, but only as an isolated token and not as part of a layer expression.

The *keyword* is one of the DRC keywords Overlap, IfOverlap, NoOverlap, AnyOverlap, PartOverlap, and AnyNoOverlap, and the *expression* is a layer expression. If the *keyword* and *expression* are given, the *expression* must be true if an object is to be selected or deselected (with the **!desel** command). The logic is shown in the table below.

`Overlap`
: True if the object is completely covered by the *expression*.

`IfOverlap`
: True if the object is completely covered or completely uncovered by the *expression*.

`NoOverlap`
: True if the object is completely uncovered by the *expression*.

`AnyOverlap`
: True if there is nonzero overlap area between the object and the *expression*.

`PartOverlap`
: True if the object is partially covered by the *expression*, i.e., not completely covered or uncovered.

`AnyNoOverlap`
: True if the object is not completely covered by the *expression*.

Examples:

```
!select l CAA b Overlap CPG
```

This will select boxes on CAA that are entirely covered by CPG.

```
!select l V1|V2 AnyNoOverlap M1 & M2
```

This will select all geometric objects on V1 and V2 that are not completely covered by both M1 and M2.

The **!select/!desel** commands with electrical property modifiers also work in physical mode. The selected cell will be the physical dual of the electrical cell containing the property. The duality must have been established with the commands in the **Extract Menu**.

Examples:

Select all instances of the cell named "andgate":

```
!select c andgate
```

Select all instances of cells with name starting with "and". The '.' is a wildcard:

```
!select c and.
```

Select resistors R1-R9:

```
!select n R[1-9]
```

Select all polygons and wires on layer M2:

```
!select l M2 w p
```

Select everything on M2

```
!select l M2
```

A blank field is taken as "all". Entering `!select` without arguments selects everything in the cell. Giving "`!select c`" selects all subcells, etc. For the layer modifier, the literal "`all`" can be used to specify all layers (hopefully there is no layer named "all"). For example, "`!select l all b`" selects boxes on all layers. This is redundant, since "." performs the same global match as "all".

There are a couple of special cases: "`!select all`" will select all geometry (not subcells) the same as "`!select l`", and "`!select .`" will select everything, the same as with no argument.

The regular expression matching may take some getting used-to. A match will be indicated if the name contains a substring of the given string, case insensitive. For example, "`!select n Lc`" would match `Lc, Vlc, IallCnt`, etc. The circumflex ('^') can be used to force matching at the start of a string, and the dollar sign ('$') forces matching at the end of a string. Thus, to match a literal, one should use the form "^*string*$".

19.22.2 The !desel Command: Deselect Objects

Syntax: `!desel` *what qualifier_or_regex* [*keyword expression*]

This is the companion to the **!select** command. The arguments are the same, however objects indicated by the arguments are deselected if selected, otherwise there is no effect.

19.22.3 The !zs Command: Zoom to Selected Objects

Syntax: `!zs`

Giving this command will change the view in the current window (the last drawing window to contain the mouse pointer) to show all selected objects. The window will zoom in or out to show all selections, plus a small margin.

19.23 Shell

19.23.1 The !shell Command: Pop Up Terminal Window

Syntax: `!shell` [*command...*]

Giving the command "**!shell**" without arguments is equivalent to giving a bare exclamation point with no following text. If a *command* is given, that command will be run in the pop-up window. This is equivalent to `!`*command*, provided that this is not also a built-in command. The use of `!shell` removes the ambiguity.

The shell which is used to execute operating system commands can be selected by the user, through the Shell variable and the **!set** command. If this is not set, the SHELL environment variable is used if set, otherwise the default "`/bin/sh`" shell is used, except under Windows where the standard "DOS box" is the default.

Under Windows, it is possible to open a Cygwin `bash` shell window instead of the brain-dead "DOS box", if Cygwin is installed. If the Shell variable or SHELL environment variable (in that precedence) contains the Windows path to the `bash.exe` file, a bash window will be used. If neither is given, and `bash.exe` resides in `/bin` or `/cygwin/bin` on the current disk drive, or the `CYGWIN_BIN` environment variable is set to the Windows path to the directory containing `bash.exe`, a bash shell will be used. Only if `bash.exe` is not found, or one of the variables specifically invokes "`cmd`", will a DOS box be used.

19.23.2 The !ssh Command: Connect to Remote System

Syntax: `!ssh` [*hostname*]

This command will pop up a terminal window that will contain an `ssh` login process to a remote host. If the *hostname* is not given with the command, it will be prompted for.

The *hostname* can actually contain additional `ssh` options if needed, and the name of the host can be in the form *user*`@`*host*, which allows logging in as *user*.

The `ssh` process will establish X forwarding to the remote system, and will automatically set the SpiceHostDisplay variable if authentication is achieved before a time out. This facilitates using *WRspice* on the remote system to perform simulations in electrical mode, from the **run** button in the side menu. The remote system must have a `wrspiced` daemon running, and the SpiceHost variable should be set to the remote host name. The X forwarding provided by the **!ssh** shell takes care of display string setting and permissions. The **!ssh** shell must remain active while *WRspice* is in use, as exiting the shell will break the connection to *WRspice* graphics.

See the description of the SpiceHostDisplay variable in E.12 for more information.

This command will work under Windows, if Cygwin is installed, along with the Cygwin OpenSSH package. The `ssh` program will be found if it resides in `/bin` or `/cygwin/bin` on the current disk, or if the CYGWIN_BIN environment variable is set to the path to the directory that contains the `ssh.exe` binary. This is the Windows path, not the path within Cygwin. *Xic* is not a Cygwin program, and knows nothing about Cygwin mount points or symbolic links.

19.24 Technology File

19.24.1 The !attrvars Command: List techfile attribute variables

Syntax: `!attrvars` [*filename*]

Most of the internally recognized variables can be set from the technology file (see A.8.9), using the same syntax as for technology file keywords. The variables are categorized as boolean or string types, which are set using different syntax forms.

Most, but not all, variables can be set in this way. There are a few that are strange in one way or another and are excluded.

This command will list, in *filename*, the boolean and string variable names that can be set in this manner. This is intended for reference purposes, and the list is rather long.

If *filename* is not specified, "`attrvars.txt`" is used.

19.24.2 The !dumpcds Command: Create Virtuoso^{TM} Startup Files

Syntax: `!dumpcds` [*basename*]

This command dumps Cadence *Virtuoso*-compatible ASCII technology, display resource (DRF), and GDSII layer mapping files based on the present *Xic* technology database. The files produced will be *basename*`.txt`, *basename*`.drf`, and *basename*`.gdsmap`, respectively. If no *basename* is given, it defaults to "`xic_tech_cds`".

19.25 Update Release

19.25.1 The !update Command: Download/Install Update

Syntax: `!update`

This command is equivalent to giving the special keyword "`:xt_pkgs`" to the help system, which brings up the *XicTools* package management page (see 6.1.1). The page lists installed and available packages for each of the *XicTools* programs for the current operating system, and provides buttons to download and install the packages.

Unlike in earlier *Xic* releases, there is no provision for automatic checking for updates, so this command or equivalent should be run periodically to check for updated packages.

19.26 Variables

19.26.1 The !set Command: Set Variables

Syntax: `!set` *name* [*value*]

The **!set** command is used to set variable *name* to *value*. The *name* is the first token following **!set**, and *value* represents the rest of the line (which may be empty). White space is stripped from the front of the first word in *value* and after the last word in *value*. If *value* is blank, the variable is understood as a boolean, and is "set".

Any variable name can be set in this manner, though there are a number of variables with predefined names which have significance to *Xic* operation, which are listed in Appendix E. Furthermore, device properties can be set with a variant of this command. A variable which has been set can be removed with the **!unset** command.

In the **!set** command, tokens in the *value* string of the form $(*setvar*) are expanded to the string associated with *setvar*, if *setvar* has been set previously. This applies if *setvar* was set with the **!set** command or related script functions, or if *setvar* is set in the environment, i.e., is an environment variable (see 2.5). If *setvar* is not resolved, no change is made. Otherwise, in general, the token is replaced with the value of *setvar*.

There is an exception to the direct-substitution rule. If any substitution string is of the form "`(...)`", then the parentheses and leading/trailing white space are stripped before substitution, and the entire substituted string is enclosed in parentheses if it is not already. This is for convenience when adding a directory to a search path (see 2.6) variable, and the path is enclosed in parentheses, when using forms like

`!set path` *dir* `$(path)`

In this case, the modified substitution rule ensures that *dir* is logically placed in front of the search path in `path`. For example, if `path` is

`( /dir1 /dir2 )`

then after the substitution implied above, one has

`path = (` *dir* `/dir1 /dir2 )`

which is correct. If the direct substitution was applied instead, this would give

`path =` *dir* `( /dir1 /dir2 )`

which is garbage as interpreted as a search path.

19.26.2 The !unset Command: Unset Variables

Syntax: `!unset` *varname*

This command will remove the previously set *varname* from the internal list of variables which have been set. Some internal variables, such as the paths, can not be unset, however they can be altered with the **!set** command.

19.26.3 The !setdump Command: Dump Variables

Syntax: `!setdump` [*filename*]

This command will dump to *filename* a listing of all of the currently defined variables, in a format accepted by the script parser, i.e., as a series of `Set` function calls. This block can be cut/pasted into an initialization file to restore state.

If the *filename* is not given, output goes to the standard output.

19.27 *WRspice* Interface

19.27.1 The !spcmd Command: Run *WRspice* Command

Syntax: `!spcmd` [*WRspice* command ...]

This will establish a stream to *WRspice* (if not already established) and run the command (if given). This is a means for running arbitrary *WRspice* commands. Text output goes to the console window.

In addition to the *WRspice* commands, the client-side directive

`send` *filename*

is available. The *filename* is that of a local SPICE input file. The file will have `.include` and `.lib` lines expanded locally, and `.spinclude`, `.splib` lines will be converted to "`include`", "`.lib`", as is done for decks created within *Xic*. The result will be sent to *WRspice* and sourced.

This page intentionally left blank.

Appendix A

Technology File

The technology file tells *Xic* all it knows about the layers and display attributes, as well as being a general source of initialization information. The name of the file is "`xic_tech`", and an extension *.xxx* can be added to the name, so that if *Xic* is started with the `-T`*xxx* option, the technology file with the extension will be used. For example, "`xic -Ttrw`" would cause *Xic* to read `xic_tech.trw`.

It is legitimate to start *Xic* without reading a technology file, by using "`xic -T`". In this case, new layers will be assigned as needed as cells are read in. This can be useful for examining an undocumented GDSII file, for example. Once the layout has been read in, new colors and fill styles can be assigned, and the **Save Tech** command in the **Attributes Menu** used to dump an appropriate technology file for the next time.

The technology file is sought first in the current directory (where *Xic* was started). If the environment variable XIC_TECH_DIR is set to a directory path, that directory is searched. If a subdirectory of the current directory named "`techfiles`" exists, it is searched next. Finally, the technology file is searched for along the library search path. The library path can be set with the environment variable XIC_LIB_PATH. The default path is

```
( .  /usr/local/xictools/xic/startup ).
```

The first matching terchnology file found in the search will be used. The default technology file has been provided by your system administrator. A personalized version can be generated with the **Save Tech** command.

The technology file generally begins with comment lines explaining the process that the file supports. The order of the sections that follow is rather flexible, though the printer driver blocks should appear last. It is recommended that one follow the ordering described here, which is the order used by *Xic* when generating a technology file, to be on the safe side. None of the sections is required to exist. Technology files for *XicII* and *Xiv* feature sets are simplified, omitting the sections that apply to unavailable features.

At the top of the file are macro definitions using the `Set` or `Define` keywords, and **!set** lines for setting global variables. The introductory part of the file further consists of optional path specifications. The layer blocks follow, which is where the core information about the particular technology resides. The electrical layers are defined first, followed by user-defined design rules, followed by the physical layer definitions.

The physical layers are followed by the standard via definitions, then the device blocks, where physical characteristics for device extraction are given. These are followed by script function definitions. Finally, there is a section containing display attribute specifiers and other parameters, and the hard-copy driver parameter blocks.

Long lines can be continued in the technology file by using backslash continuation. For example, the following would be read as one line:

```
This a line to be continued, the backslash \
must be the last character in the line.
```

The technology file has a macro facility which can be used to simplify the constructs and to customize the file to a particular variation of the technology.

The technology file may contain the following keyword/value pairs near the top of the file:

`Technology` *name*
The *name* can be any character token (no white space allowed) and defines a value for the predefined TECHNOLOGY macro. Additionally, the value of *name* is itself defined as a macro. These are not directly used by *Xic*, but the macro is placed in the name space of the macro preprocessor used when reading various types of input files, including the device library. The name is displayed in the status line of the main window, and is part of the information available for output in scripts and elsewhere.

`Vendor` *name*
The *name* can be any character token (no white space allowed) and defines a value for the predefined VENDOR macro. This is not directly used by *Xic*, but the macro is placed in the name space of the macro preprocessor used when reading various types of input files.

`Process` *name*
The *name* can be any character token (no white space allowed) and defines a value for the predefined PROCESS macro. This is not directly used by *Xic*, but the macro is placed in the name space of the macro preprocessor used when reading various types of input files.

`DeviceLibrary` *libname*
The *libname* is the name of a device library file which provides device outlines for use in schematics. If not given, the name defaults to "`device.lib`". The *libname* should be a file name, without any directory path. A file by that name should be found in the library search path on program startup.

`ModelLibrary` *libname*
The *libname* is the name of a model library file which provides SPICE models for use in SPICE output. If not given, the name defaults to "`model.lib`". A file by that name should be found in the library search path on program startup.

`ModelSubdir` *dirname*
The *dirname* is the name of a subdirectory of the directories of the library search path, in which are found SPICE model files. All directories of this name found in the library path will be searched for SPICE models. If not given, the name defaults to "`models`".

`ReadDRF` *filename*
This is part of the Cadence^{TM} compatibility package (see 5.7). The *filename* is the name of or path to a file in the format of a Virtuoso display resource file (including those from the Ciranova PyCell Studio). The full path should be given unless the file is in the library search path. This provides display attributes for layers.

`ReadCdsTech` *filename*

This is part of the Cadence^{TM} compatibility package (see 5.7). The *filename* is the name of or path to a file in the format of a Virtuoso ASCII technology file. The full path should be given unless the file is in the library search path. This provides layer and purpose definitions, rules, constraints, and other technology data. Layers defined in this file will appear in addition to those defined elsewhere.

`ReadOaTech` *library*

This will obtain Virtuoso technology information directly from OpenAccess. The *library* is an OpenAccess library, listed in the `lib.defs` or `cds.lib` file. This obtains technology information by use of the OpenAccess plug-in. There should be no reason to use both this and `ReadCdsTech`, as they should retrieve the same information.

`ReadCdsLmap` *filename*

This is part of the Cadence^{TM} compatibility package (see 5.7). The *filename* is a path to a Virtuoso layer-mapping file, which provides GDSII layer/datatype numbers for the layers. This can be used in addition to, and must be called after, `ReadCdsTech`. It is used to import the Stream mapping for the layers.

`ReadCniTech` *filename*

The *filename* is the name of or path to a file in the format of a Ciranova (now Synopsys) ASCII technology file. The full path should be given unless the file is in the library search path. This file format is superficially similar to a Virtuoso ASCII technology file, yet sufficiently different that a separate reader is required. The format is documented in the PyCell Studio distribution from Synopsys, and example files are provided (see 5.6).

When setting up a technology file for the PyCell Studio or something similar using Ciranova technology, it may be necessary to use this keyword more than once, if the technology is described in more than one file. It is also necessary to use the `ReadDRF` keyword to read display resource files.

For example, here is a skeletal technology file for the Ciranova 130nm model process in the PyCell Studio, which is installed under `/usr/local/ciranova`.

```
Set cni130 = /usr/local/ciranova/tech/cni130
ReadDRF $(cni130)/santanaDisplay/SantanaDisplay.drf
ReadCniTech $(cni130)/santanaTech/Santana.tech
ReadCniTech $(cni130)/santanaDisplay/SantanaDisplay.tech
```

The ability to read the Lisp/Skill file format used by Virtuoso is provided by an internal Lisp parser. The parser is available to run general scripts through the **!lisp** command, though this has limited utility at present.

In the technology file, is is sometimes useful to enable debugging output from the Lisp parser. The following keyword enables this.

`LispLogging` [`y/n`]

If this boolean keyword is set in the technology file, a log file will be generated when the Lisp parser is used. This can be used to track down issues when parsing Virtuoso-style input files. Asserting this keyword is equivalent to setting the Lisp logging in the **Logging Options** panel from the **Help Menu**, which otherwise can't be done before the technology file is read on program startup.

The logging output is put into a file named *filename*`-lisp.log` in the logfiles directory. The *filename* is the name of the input file being parsed.

A.1 Technology File Comments

The technology file recognizes a `Comment` keyword. These lines have no effect, but are saved and included when the file is written with the **Save Tech** command. Thus, notes about the file can be preserved. An attempt is made to to place the comment in the same relative position during an update.

Comments can also be included in the technology file after the '`#`' character or '`//`' sequence, however these comments will not appear in a file written with the **Save Tech** command.

Example:

```
Comment Technology file for the Ultra-MOS version 3.5 process
Comment Version 1.3 March 24, 2002 George H. Frump
```

A.2 Technology File Macros

In order to facilitate customization of the technology file to different variations, in particular to support scalable technology, a macro facility is provided, along with an expression evaluator. Macros can be used to simplify or clarify the constructs used in the technology file, and facilitate portability by effectively customizing the technology file to different environments.

The macro capability makes use of the generic macro preprocessor provided in *Xic*, which is described in 18.1. The reader should refer to this section for a full description of the preprocessor capabilities. The preprocessor provides a few predefined macros used for testing (and customizing for) release number, operating system, etc. The keyword names, which correspond to the generic names as described for the macro preprocessor, are case-insensitive and listed in the following table.

Keyword	**Function**
`Define`	Define a macro.
`If`	Conditional evaluated test.
`IfDef`	Conditional definition test.
`IfnDef`	Conditional non-definition test.
`Else`	Conditional else clause.
`Endif`	Conditional end clause.

A macro definition can appear anywhere in the technology file. Throughout the technology file, each line is macro expanded. The actual arguments replace the formal arguments (if any) in the substitution text, which replaces the macro reference. The macro is recognized as a text token.

Example:

```
Define mytext(x) this is rule number x
...
MinWidth 2 # mytext(1.2)
```

The `MinWidth` line expands to

```
MinWidth 2 # this is rule number 1.2
```

The conditional keywords provide tests which can be used to select which lines of the technology file are actually read, based of the settings of existing macros and/or expression evaluation. The logic is explained in the description of the generic macro preprocessor.

Example:

```
Define TightRules
...
Layer M1
IfDef TightRules
MinWidth .4
Else
MinWidth .8
Endif
```

In the example above, commenting out the `Define` line

```
#Define TightRules
```

reconfigures the technology file.

When the technology file is updated with the **Save Tech** command, only the lines that were actually processed are written, i.e., the `IfDef`, etc. lines and unused blocks are stripped.

A.2.1 The `Set` Keyword: Variable Expansion

A different type of macro is defined using the `Set` keyword, where the words following are parsed into three tokens

`Set` *name* `=` *value*

This type of macro is referred to by

`$(`*name*`)`

which is replaced by *value* as the file is read. If the *name* has not been assigned in a `Set` line, but an environment variable by that name is found, the substitution will be made from the value of the environment variable. Otherwise, the variable must be set before being referenced, meaning that the `Set` line must appear before the first reference in the technology file.

Neither the *name* or *value* tokens can contain a carriage return, though they can contain embedded white space. In either case, the beginning and end of the token is the first and last non-white character. Substitution is performed recursively. The two types of macro can be mixed, though the `Set` line is not expanded for `Define`'ed macros. Other lines are first expanded for `Define`'ed macros, then for `Set` macros.

The `Set` keyword should not be confused with the **!set** command, which can also appear in the technology file.

A.2.2 The `eval` Keyword: Expression Evaluation

An expression involving integers or floating point numbers can be evaluated as the file is read, with the result inserted into the line at the place of evaluation. This facilitates, for example, the use of design rules based on the *lambda* concept. In this type of rule set, design rules are specified in terms of a minimum dimension *lambda*. The *lambda* may vary between different process implementations. In the technology file, *lambda* is defined as a macro, and inputs to the design rule specifications is evaluated in terms of *lambda*.

The syntax for expression evaluation is `eval(`*expression*`)`. This construct can occur anywhere in the text, although it makes sense only where a number is expected. The result of the evaluation is substituted into the text replacing the `eval` construct, before that line of the technology file is interpreted. The expression is interpreted by the parser otherwise used for interpreting command scripts, and the full complement of operations and functions is available. Macros are expanded before the expression is parsed.

Example:

```
Set lambda = .6
...
PhysLayer BASE
MinWidth eval(2*$(lambda)) #Minimum width of the BASE layer is 2*lambda
```

In this example, the parameter `lambda` is defined to ".6" with the `Set` keyword. Elsewhere in the file, design rules can be specified as functions of `lambda` using the `eval` construct, as shown.

Example:

```
Set lambda = .6
Define L(x) eval($(lambda)*x)
...
PhysLayer BASE
MinWidth L(2) #Min width of BASE layer is L(2)
```

In this example, the macro `L(x)` is used to hide the call to the evaluation function, simplifying syntax.

If the technology file is updated to disk using the **Save Tech** command button, only the macros used in the design rule keywords will be preserved in their original macro form in the new file. Elsewhere, the written lines will contain the expanded quantity. All of the `Set` and `Define` lines will be preserved. Thus, the use of macros should be restricted to the design rule keywords, unless the user is willing to hand edit the new files produced with the **Save Tech** command.

A.3 Technology File Global Variables

Also typically appearing near the top of the technology file are the `!set` commands.

`!set` *arguments*

Unlike the `Set` keyword, this directive assigns variables as if the keyboard **!set** command, as used interactively from the prompt line, had been given. The *arguments* are exactly as they would appear on the prompt line. Thus, the command attributes that are controlled with the **!set** command can be specified in the technology file. The technology file is read after the `.xicinit` file and before the `.xicstart` initialization file, which are other options for executing the **!set** command at program startup.

This form is appropriate for variables that are defined by the user. Variables that are known to *Xic* can presently be set as keywords (see A.8.9), though the form described here can be used as well.

When a new technology file is written with the **Save Tech** command, all **!set** lines from the original technology file (if any) are written as a block, but commented out. This is followed by another block containing all of the currently defined variables, except for those known to *Xic* that can be set as keywords. These include the path variables, and are written as keyword definitions elsewhere in the file. The present list will contain variables defined by the user. These lines are active. The user can edit these blocks as necessary.

The **!attrvars** command generates a listing of the variables that can be set as technology file keywords.

A.4 Technology File Path Definitions

There are four search paths that may be specified. In each case, the path specification consists of a keyword, followed by the path. The format of the path is described in section 2.6 detailing the *Xic* search paths.

In the path defaults below, if the XT_PREFIX environment variable is defined, its value will replace "`/usr/local`".

`Path` *path*
: The `Path` keyword specifies the path to design data files: native cell, archive, and library files. The current directory "." should generally be listed first in this path. The design data path can also be set in the environment with the `XIC_SYM_PATH` variable. A specification in the technology file will override a specification in the environment.
Default: `( . )`

`LibPath` *path*
: The `LibPath` keyword specifies the path to the startup files. The startup files include the device library (default name `device.lib`), and the model library (default name `model.lib`). This path can also be set with the environment variable XIC_LIB_PATH, and a specification in the technology file will override an environment specification. Unlike other search paths, the current directory is always checked first when looking for files in this path, as if '.' was the first component.
Default: `( . /usr/local/xictools/xic/startup )`

`HlpPath` *path*
: The `HlpPath` lists directories containing database files for the help system. These files have names with suffix `.hlp`, and it is possible for users to create customized help files for their own purposes (the format is described in C.3). The help path can also be specified with the environment variable XIC_HLP_PATH, which will be overridden by a specification in the technology file.
Default: `( /usr/local/xictools/xic/help )`

`ScriptPath` *path*
: The `ScriptPath` contains directories where *Xic* searches for user generated command scripts. The script files have names with suffix "`.scr`", except for the library script which is named "`library`". This path can also be set with the environment variable XIC_SCR_PATH, which will be overridden by a specification in the technology file.
Default:`( /usr/local/xictools/xic/scripts )`.

Note that the XIC_LIB_PATH variable can be used to define the location of the technology file, and then redefined in the technology file to provide alternate locations for the device and model library files.

The path keywords, and all other keywords, are interpreted without case sensitivity when the technology file is read.

A.5 Technology File Scripts

Scripts can be included in the technology file. These scripts can appear as buttons in the **User Menu**, as with other scripts, or they can be "run once" scripts. This feature is useful for including simple technology-specific commands, such as those that create special extraction layers or physical features. Scripts defined in the technology file, however, can not be loaded into the debugger.

A script is included In the technology file as follows. The `Script` keyword is followed by the text which will appear in the command button. If the button text contains white space, it must be quoted, e.g.,

```
Script "My Cell Counter"
```

The lines of the script follow, and the script text must be terminated with the keyword `EndScript` on a separate line.

`Script` *menu_label*
script text
...
`EndScript`

If the line

```
RunScript
```

appears anywhere after the `Script` line and before `EndScript`, the script is taken as a "run once" script. It will not be added to the **User Menu**. Instead, it will be executed after the technology file has been read, then discarded. Any number of scripts can be treated this way, they execute in order of appearance in the technology file.

Scripts defined in the technology file have lower priority than other scripts in the event of a menu label text clash. Thus, technology file scripts will be "hidden" by other scripts with the same menu label, should any exist.

A.6 Technology File Layer Blocks

Xic maintains a table of layer aliases, which can be used instead of the actual layer name where a layer entry is required. This follows the Virtuoso "`techParams`" definitions where the value is a layer name. The alias name is intended to be a generic name such as "`active_layer`", or "`nwell_layer`", which can be used in device blocks and elsewhere to provide a degree of process independence. Further, some of these names may be specific to Virtuoso, and be handled in special ways. The only example of this at present is handling of "`active_layer`".

Each line of the layer alias list takes the following form.

`MapLayer` *alias layer_name*

The *layer_name* must be resolvable as an *Xic* layer.

`active_layer` alias handling

If a layer alias named "`active_layer`" is given, as is at least one of the alias names "`ngate_layer`" and "`pgate_layer`", and the `active_layer` does not have a `Conductor Exclude` directive, one will be created. The excluded area is logically `ngate_layer|pgate_layer`. This supports correct MOS device recognition when technology data are obtained exclusively from Virtuoso.

The component layer names and numbers, and purpose names and numbers, are specified in optional tables. These tables must appear before any *Xic* layer definitions. A layer name or purpose name used by an *Xic* layer that is not found in a table will be created, and assigned a number by *Xic*. The tables ensure a strict and repeatable correspondence between names and numbers, which may be necessary for compatibility with other tools.

The tables consist of lines in the following form:

`DefineLayer` *layer_name layer_number*
`DefinePurpose` *purpose_name purpose_number*

Name strings may contain alphanumeric characters plus the dollar sign ('`$`') and underscore ('`_`'). The numbers can be any value representable with 32-bits, except that -1 is reserved. Be aware that other tools may define ranges of values that are reserved for internal use.

Following the layer and purpose tables, if any, *Xic* layers may be defined. There are separate definitions of layers used in electrical (schematic) mode, and in physical mode (for layouts). *Xic* maintains a standard set of electrical layers, in a standard order. These will be created if the definitions do not appear in the technology file (or no technology file is read). The SCED layer, which is the electrically-active wiring layer, is always first. The user can modify the presentation attributes, and add layers as desired. For physical mode, there are no such layers, all layers must be defined in some manner.

The separation of electrical and physical layers is a bit of an anachronism, and in current *Xic* releases a user-defined layer can actually exist in both electrical and physical layer tables. This accommodates technologies imported from other tools, such as Cadence Virtuoso, where no such distinction is made.

Each layer definition starts with the keyword `PhysLayer` for physical layers or `ElecLayer` for electrical layers, followed by a name. Both of these keywords have synonyms (listed below) for backwards compatibility. The name should be a valid layer name, though an attempt is made to use invalid names if possible by editing out unacceptable characters.

Layer blocks appear in a contiguous section in the technology file, and in physical mode will appear in the layer table in the order given. In electrical mode, reordering may be applied, as there are some internal assumptions.

A layer block is terminated by the start of another layer block, or by a keyword which would logically end per-layer parsing.

`ElecLayer` *name*

This keyword specifies the beginning of the layer block for the electrical layer *name*. The keyword `ElecLayerName` is a synonym.

`PhysLayer` *name*
This keyword specifies the beginning of the layer block for the physical layer *name*. Layers will appear in the physical mode layer table in the order given. The keywords `PhysLayerName`, `Layer`, and `LayerName` are all synonyms for this keyword.

`DerivedLayer` *name* [`join|split|splitv`] *expression*
This line provides a definition of a derived layer. Derived layers represent an expression of other layers, derived and normal physical, which can be referenced in layer expressions. Derived layers were introduced in support of the design rule checking system, but can be accessed for other purposes through a script function interface.

This will add a derived layer to the database, under the name given in the first token. The remainder of the line is the layer expresion. The expression is not parsed until evaluation time.

When the derived layer is evaluated, the geometry can appear as an assemblage of trapezoids if either of the `split` or `splitv` keywords is given, or alternatively as a minimal number of complex polygons if the `join` keyword is given instead. If `splitv` is given, a vertical orientation is favored for the decomposition, whereas similarly `split` will produce a decomposition favoring a horizontal orientation. The default is the joined form if none of these optional keywords is given, except when simply copying from another layer in which case the default is to copy objects without change. The keyword "`splith`" is a synonym for "`split`".

These lines begin a layer block description, and any of the keywords which can apply to physical layers can be used in the derived layer blocks, though the definitions may be useless. Layer block keywords that are significant are listed below.

1. Design rules. These rules will be evaluated while doing design rule checking. As further described in the in the DRC description, there are some types of tests that require use of derived layers.
2. When a new normal layer is created as a copy of a derived layer, which can be done with the **!layer** command, or with the **Layer Expression Evaluation** panel from the **Edit Menu**, or with the `Layer` script function, the new layer will inherit the attributes of the derived layer. This includes color, fill pattern, GDSII mpping, and other flags and properties. This gives purpose to the definitions provided in the derived layer block.

The sub-sections that follow categorize and describe the fairly lengthly list of per-layer keywords. All of the keywords are optional, and can appear under an electrical or physical layer, unless stated otherwise. Many of these keywords can be programmed from within *Xic* with the **Tech Parameter Editor** from the **Attributes Menu**. Other panels from the **Attributes Menu** allow setting colors, fill patterns, etc. which correspond to values from keywords.

In the syntax descriptions, the italicized quantities represent data the needs to be provided. The "`y|n`" symbol implies that one of '`y`' or '`n`' should follow the keyword. Actually, '`0`' (zero), or any word that begins with the letters or sequence (case insensitive) '`n`', '`f`', '`of`' is taken as a false value. Anything else, including no following text, is taken as true ('`y`' is always redundant).

A.6.1 Technology File Layer Block Keywords: Misc. Attributes

These miscellaneous keywords apply bits of information to the layer, which affects behavior in situations described.

`LppName` *name*
This provides an optional alias for the layer/purpose pair that represents the *Xic* layer name. The

Xic layer can be accessed by this alias, in addition to the normal name. If no non-space characters are found after the keyword, the statement is ignored. Any character is allowed in the alias name, but leading and training white space is removed, and inclusion of some characters, for example a colon (':'), can definitely cause trouble.

`Description` *description_string*
: This will set the description field of the current layer. If no non-space characters are found after the keyword, the statement is ignored. Leading and training white space is removed from the description string.

`NoSelect` [y|n]
: If this keyword appears, and any following argument indicates true, objects on the layer can not be selected. The selectability status of the layers can be changed from the layer table.

`NoMerge` [y|n]
: This keyword indicates that automatic merging of objects is suppressed on the layer. This overrides any merging enabled by the **Merge new boxes and polys with existing boxes/polys** and **Clip and merge new boxes only, not polys** check boxes in the **Editing Setup** panel from the **Edit Menu**, and the **Clip and merge overlapping boxes** button in the **Set Import Parameters** panel from the **Convert Menu**, and the corresponding variables.

`WireActive` [y|n]
: If this keyword appears, and any following argument indicates true, wires on the layer will be considered for wire connectivity in schematics. This flag is always set implicitly in the `SCED` layer. The Cadence compatibility system may create a layer named `wire` with purpose `drawing` which will have this flag set.

`Symbolic` [y|n]
: This keyword indicates that the layer will not be shown in the display produced by the **Cross Section** command (in the **View Menu**). Otherwise, it doesn't have any purpose in *Xic*, but might be useful to the user as a flag to indicate a non-physical layer.

`Invalid` [y|n]
: If this keyword appears, and any following argument indicates true, the layer will not appear in the layer table, but will exist internally and resolve any references to the layer in a design. Such layers are invisible, as the redisplay involves cycling through layers in the layer table.

  This is for compatibility with Cadence Virtuoso, whose layer presentation attributes include a `Valid` flag. When reading a Virtuoso technology file, if a layer is invisible, not selectable, and is invalid, the *Xic* `Invalid` flag will be set.

A.6.2 Technology File Layer Block Keywords: Presentation

These keywords impact the appearance of objects on the layer on-screen and in prints.

`RGB` *colorspec*
: This keyword will set the color used to render objects on the layer on-screen. The *colorspec* string is the name of a color or an RGB triple:

  - The name of a color. The recognized names can be listed from the **Set Color** pop-up in the **Attributes** menu with the **Colors** button.
  - Three space-separated numbers, each 0–255, representing the red, green, and blue intensity. E.g., "`196 240 235`".

- Other forms recognized by the `XParseColor` C library function, including "`#RRRRGGGGBBBB`" and "`rgb:RRRR/GGGG/BBBB`". Here, `R`, `G`, and `B` are single hexadecimal digits.

If the color is given as a name, the color will be converted to its RGB values if the file is updated. If no `RGB` keyword is given for a layer, *Xic* will assign a random color. The `RGB` keyword is allowed in the mini-layer blocks found in the print driver specifications.

`Filled` [y[...]]
`Filled` `n`[...] [`o, f, c`]
`Filled` *bit_data* [`o, f, c`]

This keyword sets the fill and outline style used to render objects on the layer. The tokens (other than *bit_data*) can be words starting with the indicated letters, or or just the letters themselves, e.g., "`n`", "`no`", and "`none`", are equivalent. This is case-insensitive.

If no tokens follow the keyword, or the first token starts with '`y`', solid fill will be used. Additional tokens on the line will be ignored.

If the first token starts with '`n`', no fill pattern (empty fill) will be used. In this case, there are three outline styles available:

1. A thin solid line boundary.
2. A thin dashed line boundary.
3. A thick solid line boundary for Manhattan boxes and polygons, and a thin solid line boundary for other objects.

There is also the "cut" attribute, where diagonal lines are drawn over boxes, forming an X. This applies to boxes only, not wires or polygons, even though they may be rendered as four-sided rectangular figures.

Any text that follows the word that started with '`n`' is examined for the presence of the characters '`o`', '`f`', and '`c`'. These can be found as individual letters or parts of words, for example "`outline cut`" and "`oc`" and "`o c`" are all equivalent. In addition, this is all case-insensitive.

If neither '`o`' or '`f`' is found, a thin solid outline (style 1) is used. If '`o`' is found but not '`f`', a thin dashed line (style 2) is used. If '`f`' is found, with or without '`o`', then a thick solid line is used for edge segments of Manhattan objects, and a thin solid line is used for non-Manhattan objects (style 3).

In any case, if '`c`' is found, the "cut" attribute is applied. If '`o`' is also found buf not '`f`', the diagonals are shown as dashed lines, the same as the boundary. Otherwise, the diagonals are always thin solid lines.

The form on the third line is used to specify a stipple pattern to use for fill. *Xic* supports any stipple map size with the x and y dimensions in the range of 2–32. However, *Xic* releases prior to 3.2.25 supported only 8x8, 8x16, 16x8, and 16x16 maps. The format described here is generally not backwards compatible with these releases.

Maps can be read as hex numbers, or as ASCII tokens, but not in the same line. When *Xic* writes a technology file, the default is to use the ASCII token format, which actually renders the map in a crude way. This format is best illustrated by an example:

```
Filled \
  |   ..   |  (0x18) \
  |  ....  |  (0x3c) \
  | ...... |  (0x7e) \
  |...  ...|  (0xe7) \
  |...  ...|  (0xe7) \
```

```
| ...... |  (0x7e) \
|  ....  |  (0x3c) \
|   ..   |  (0x18) outline
```

The points to note here are the following.

1. Line continuation is used so that the map is visible to a human reader. This is not required in general.
2. Each line of the map contains space and non-space characters, surrounded by '|' characters. Although a period is used here, any non-space printing character will work.
3. Each of these must contain the same number of characters, this number being in the range 2–32. This sets the width of the map.
4. The number of these constructs found in the line sets the height of the map. This must be in the range 2–32.
5. The map data parser ignores anything enclosed in parentheses. Above, the equivalent hex number for the data pattern is provided, but is ignored by the parser.

An equivalent form using hex data is

[x=*width*] [y=*height*] *hex_number hex_number ...*

The *width* and *height* are decimal numbers in the range 2–32. The number of hex digits that follow must match the *height*.

The width and height specifications can be omitted, in which case the format reverts to the pre-3.2.25 expectation. The hex numbers must be one of

- 8 2-digit hex numbers that specify an 8x8 map.
- 16 2-digit hex numbers that specify an 8x16 map.
- 8 4-digit hex numbers that specify a 16x8 map.
- 16 4-digit hex numbers that specify a 16x16 map.

Additional text on the line is examined for the 'o', 'f', and 'c' characters as described above for the no-fill case. With a fill pattern, the interpretation is slightly different, as there is no dashed line outline available in this case. If neither 'o' or 'f' appear, the pattern will not be outlined. If 'o' appears without 'f', a thin solid outline will be used. If 'f' appears, edges of boxes and Manhattan polygons will be thick. The 'c' will draw diagonals on boxes. For historical reasons, the character 'y' is treated the same as 'o'.

If the boolean variable TechNoPrintPatMap is set when *Xic* writes a technology file, then the hex form will be used to specify fill patterns. Otherwise, the ASCII form is used.

Here are a few more example fill specifications:

```
Filled y
Filled no fat
Filled cc aa cc aa cc aa cc aa outline
```

In electrical mode, the SCED layer defaults to solid fill, and other layers default to empty fill with a thin outline. All layers default to empty fill with a thin outline in physical mode. The `Filled` keyword is allowed in the mini-layer blocks found in the print driver specifications.

`Invisible` [y|n]
If this keyword appears, and the following argument indicates true, the layer will not be visible, though it will appear in the layer table, where the visibility status can be changed.

The `Invisible` keyword is allowed in the mini-layer blocks found in the print driver specifications. This is the only place where use of the `y|n` argument may be needed, in particular if `Invisible` is specified in the main layer block, `Invisible n` may be used in the driver block to make the layer visible in print driver output.

`Blink` [y|n]
If this keyword appears, the layer color will oscillate between two shades with a 0.5 second period. This is only supported in pseudo-color (usually 256 colors) graphics mode.
Default: not blinking

`NoInstView` [y|n]
If this keyword appears, and any following argument indicates true, objects on the layer will not be shown in electrical instances of the containing cell. However they will appear when the cell is the current cell. This is ignored in physical node.

`WireWidth` *width*
This keyword can appear in physical layer fields. The *width* is a floating point number which sets the default wire width to that value in microns. This value will be used when wires are created in *Xic*.
Default: 0

CrossThick *thickness*
This keyword, which can be applied to physical layers only, sets the layer thickness as rendered in the **Cross Section** command in the **View Menu**. The *thickness* is given in microns.

A.6.3 Technology File Layer Block Keywords: Conversion

The following keywords set the layer mapping for GDSII and OASIS format input and output. These can be programmed from within *Xic* with the **Tech Parameter Editor** in the **Attributes Menu**.

`StreamData` *layernum datatype*
This keyword is deprecated, and can be read but is not generated by *Xic*. The *layernum* and *datatype* are the layer mapping used when converting to and from GDSII format. The layer must be in the range 0 through 65535, and the datatype can take values -1 through 65535. Values larger than 255 are outside of the GDSII specification, but are sometimes used anyway although files containing such data may not be generally portable. If -1 is given as the datatype, all GDSII datatypes will be mapped to the present *Xic* layer, and datatype 0 will be used for output. Otherwise, the layer and datatype in a GDSII file must match those given for successful mapping to the *Xic* layer. Note that often the end of range values are reserved in other CAD environments, and that some releases of the GDSII format support only 64 layers and datatypes. The datatype is used by *Xic* only in conjunction with the `NoDrcDatatype` keyword, and is otherwise typically set to 0. This keyword has been superseded by `StreamIn` and `StreamOut`.

StreamIn *layer_list* [, *datatype_list*]
This keyword specifies a set of layer/datatype combinations that will map to the present *Xic* layer when reading GDSII and OASIS files. Any number of such lines can be present. The *layer_list* is a space-separated list of tokens, each of which is either a GDSII layer number ("32") or a range of numbers ("35-41"). The *datatype_list* is similarly constructed, and is optional. The numbers in

either list can range from 0 to 65535, though numbers larger than 255 are outside of the GDSII specification (but sometimes used anyway). If a *datatype_list* appears, it is separated from the *layer_list* with a comma. The line specifies that each of the datatypes listed on each of the GDSII layers listed will be converted to the present *Xic* layer. If the datatype list is absent, it defaults to "0-65535". For example,

```
StreamIn 5 7 8 21-30, 0 20-63
```

specifies that datatypes 0 and 20-63 on GDSII layers 5, 7, 8, and 21-30 will be mapped to the present *Xic* layer as a GDSII or OASIS file is read. Note that GDSII layers cam be mapped to more than one *Xic* layer. In this case, the geometry will be created on each of the *Xic* layers mapped to.

It is possible for more than one *Xic* layer to map from a given GDSII layer/datatype. If the MultiMapOk variable is set, then multiple objects will be created when a GDSII or OASIS file is read, one on each matching *Xic* layer. If this variable is not set, only the first mapping will be used, which will be the lowest matching layer found in the layer table.

StreamOut *out_layer* [*out_datatype*]

This line specifies a layer/datatype combination to be used when generating GDSII and OASIS files for the present *Xic* layer. One of these should appear for each *Xic* layer. The *out_layer* and *out_datatype* can be in the range 0–65535, though numbers larger than 255 are outside of the GDSII specification but are sometimes used anyway. Be aware that use of numbers larger than 255 may render the file non-portable. Note that often the end of range values are reserved in other CAD environments, and that some releases of the GDSII format support only 64 layers and datatypes. The default datatype, if not given, is 0.

If there are more than one StreamOut lines given for a layer, and the MultiMapOk variable is set, the objects will be added to the GDSII or OASIS file on each of the GDSII layers/datatypes specified. If the variable is not set, only the first StreamOut specification will be used.

There is no default for this keyword.

NoDrcDatatype *datatype*

If this keyword is given, then any object that has the given datatype will be ignored during DRC. On output, objects that have their DRC skip flags set will be written with this datatype, and not the default datatype given in the StreamOut line. The given datatype should appear in the input mapping for the layer.

A.6.4 Technology File Layer Block Keywords: Extraction

This section describes the keyword entries which appear in layer blocks which categorize the purpose of the layer for extraction. These define the conductor layers which are involved in grouping, identify vias between conductors, etc. These keywords can appear only in physical layer fields.

All of these settings can be entered with the **Edit Tech Params** command in the **Attributes Menu** and then written to disk with the **Save Tech** command in the **Attributes Menu**, or be entered with a text editor directly into the technology file.

Some of the keywords below use layer expressions, as were described in 15.1. A layer expression in its simplest form is a layer name. More generally, it consists of an expression involving layer names, the intersection operator (`&`), the union operator (`|`), and the inversion operator (`!`). Parentheses can be used to enforce precedence. These are the same type of expressions as used in the DRC tests. The expression is "true" at points where the expression would return opacity.

Conductor [Exclude *expression*]

This keyword indicates that the present layer is to be included in conductor net grouping. If the keyword Exclude and a following layer expression are given, the regions of the current layer under which the expression is true are clipped out for grouping purposes. For example, in CMOS technology a transistor is formed by a strip of CAA (active area) bisected by a CPG (polysilicon) gate. If "`Conductor Exclude CPG`" is given in the CAA layer block, the two pieces of CAA will be given separate group numbers, which is necessary to keep the transistor source and drain separate.

Routing [it route params]

This keyword implies that the layer is a conductor used for connecting between cells. The Conductor keyword is implied, so that the Conductor keyword does not also have to be supplied, unless there is an Exclude directive. Only layers with the Routing keyword given will be considered by the extraction system for connecting between cells, and cell formal terminals will only be assigned to Routing layers. This is not absolute, however. The extraction system will place formal terminals on Conductor layers under some circumstances, if necessary.

Optionally, routing parameter definitions may follow the keyword. These provide information to a third-party auto-route system, The parameters are saved in the *Xic* technology database, and are used when writing a technology file, but are not otherwise used directly by *Xic*. The recognized routing parameter definitions are listed below. These can appear in any order. These parameters will be parsed and set when reading the technology file, but can also be set when reading Cadence ASCII technology files.

`dir=H|V|X|Y`[...]

This sets the preferred direction of routes on the layer. The "`dir=`" is literal, and is followed by a letter or word, only the first letter of which is significant. If the first letter is `H` or `X` (case insensitive), the route direction is horizontal. If the letter is `V` or `Y`, also case insensitive) the routing direction is vertical. Otherwise, an error ensues.

`p[itch]=`*px*[`,`*py*]

This provides the values for the route pitch. Only the first letter of the "`pitch`" keyword need be present. This is followed by an equal sign ("`=`"), and one or two real numbers. The numbers are pitch values in microns. If there are two numbers, the first is the horizontal pitch, the second vertical, separated by a comma. Two numbers are required only if the horizontal and vertical pitch values differ.

`o[ffset]=`*ox*[`,`*oy*]

This provides values for the route offset, and is parsed the same way as the pitch. The values are real numbers giving the offset in microns. The second number can be omitted if it is the same as the first. The offset is the routing grid origin relative to the cell origin.

`w[idth]=`*w*

This specifies the line width, in microns, used for routing. Presently, only one number is accepted, implying that horizontal and vertical routes have the same width.

`maxd[ist]=`*d*

This provides a maximum route length, in microns. A router may use this value to limit route lengths.

GroundPlane
GroundPlaneDark (alias)

This keyword indicates that the present layer is to be treated as a clear-field ground plane. The layer is given the Conductor attribute. If the keyword "`Global`" appears, then every object on the layer will be assigned to the ground group 0. This would be appropriate if the layer represents a diffusion rather than a metallic ground plane. The default is to treat this level as a normal conductor, except that when this layer is grouped in the top-level cell, the group with the largest area is assigned to the ground group.

If "`Global`" is given, the GroundPlaneGlobal variable, which activates the mode, will be set.

Only one of the ground plane keywords can appear in the technology file. Conductor group 0 is used only if a ground plane has been specified. The ground plane layer can be referenced in Via and Contact lines just as any Conductor.

GroundPlaneClear [MultiNet [0|1|2]]
TermDefault [MultiNet [0|1|2]] (alias)

This keyword indicates that the present layer is to be treated as a dark-field ground plane. These keywords imply DarkField. Giving GroundPlane (or GroundPlaneDark) and DarkField is equivalent to GroundPlaneClear without MultiNet.

Only one of the ground plane keywords can appear in the technology file. Conductor group 0 is used only if a ground plane has been specified.

Without the MultiNet keyword, connections to this layer (as specified with the Via and Contact keywords), where this layer does *not* appear, are considered as connections to ground (group 0). Although this approach may work for simple cells, it can lead to trouble. Suppose that an island of ground plane metal is used as part of the metalization for the chip pads. This would appear as a hole in the displayed representation of the ground plane layer. Then each pad will be extracted as shorted to ground!

There is provision for more intelligent handling of the GroundPlaneClear layer, allowing the layer to be included in paths and groups. If the MultiNet keyword appears, the inverse of the layer is computed, and that (temporary) layer is used in the grouping. However, it can take quite a lot of behind-the-scenes computation if the GroundPlaneClear layer has complex patterning. Inversion is also done if the **!set** variable GroundPlaneMulti is given (note: this variable was formerly named HandleTermDefault). The temporary layer is treated as a clear-field ground plane, and all references to the ground plane will be applied to the temporary layer during grouping and extraction.

The name of the internal layer created is "`$GPI`". By default, this layer is invisible. It should not be directly edited by the user. The inverse layer is an internal layer and is never written to a file during conversion or a save. During extraction the GroundPlaneClear layer is ignored, and the inverse, which is a Conductor, is used to establish connectivity.

To establish connectivity for the commands in the **Extract Menu**, the inverse layer is created according to one of the algorithms described below. An optional integer 0–2 may follow the MultiNet keyword, which indicates the algorithm used for inversion. The algorithm can also be selected by setting the variable GroundPlaneMethod to an integer in the same range, with the **!set** command.

0 The inverted layer is created for each cell in the hierarchy by computing

`$GPI = !`*GP* `& !$$`

i.e., for each cell the ground plane is inverted and the areas over subcells are removed (recall that "`$$`" is a pseudo-layer representing subcell boundaries). This is the default.

1 The inverted layer is created only in the top cell in the hierarchy, and is the inverse of a flat representation of the ground plane layer from all cells in the hierarchy. The extraction algorithm will add virtual contacts from this layer to the appropriate places in the subcells.

2 The inverted layer is created in each cell of the hierarchy by creating a flat inverse of all of the ground plane found in the cell or lower in the hierarchy.

The default (0) method is the most efficient computationally, but the method will probably fail if sibling subcells overlap. In general, it is good practice to avoid cell overlap.

Method 1 will work if subcells overlap. However, since there is no local ground plane in the subcells, generating a netlist while in a **Push** (subedit) will not yield correct results.

Method 2 is the least efficient computationally, but each cell has a local ground plane.

Via *layer1 layer2* [*expression*]

This keyword indicates that the present layer may provide connection points between conductor nets on *layer1* and *layer2*. The *layer1* and *layer2* are names of layers each of which have the Conductor, Routing, or one of the GroundPlane keywords specified. In extraction, it is assumed that the via is formed by dark area on the present layer, and vias are completely covered by *layer1* and *layer2*. A connection is indicated if the *expression* (which is a layer expression) is true at any point within the via. The Via keyword implicitly assigns DarkField. The recognition logic is as follows:

> **for each** *region* of the Via layer {
> **if** (there exists an object on *layer1* that overlaps *region*)
>   **if** (there exists an object on *layer2* that overlaps *region*)
>   **if** (there is no *expression*, **or** the area where *expression* is true in *region* is nonzero)
>   **then** the via indicates a connection between the two objects
> }

If the *expression* is not given, it is always taken as "true".

Examples:

`Via M1 M2 !RES`
A via is indicated if part of the via object on the present layer which is being evaluated is not covered by objects on `RES`.

`Via M1 M2 I2`
A via is indicated if the via object on the present layer is partially or completely covered with `I2`.

`Via M1 M2 (!I2)&(!RES)`
A via is indicated if part of the via object is not covered by `I2` or by `RES`.

ViaCut *expression*

This is applied to an insulating layer with positive Thickness given, and defines cuts through the layer from the layer expression. This applies only when using three-dimensional processing such as for cross sections and the FastCap/FastHenry interface. It allows one to separate abstract (Thickness¿ not given or zero) Via layers from the physical layers that represent dielectrics. The abstract layers are used for netlisting, LVS, etc. The patterning is applied to the dielectric ViaCut layers when computing 3D geometry. This allows abstract vias to have cuts through multiple dielectric layers, which is required for some complex layer sequences.

Example
Assume one has the following sequence of layers:

```
M1 I1A R1 I1B M2 VM12 VR12
```

The `I1A` and `I1B` layers are dielectrics, which encapsulate a resistor `R1`. `VM12` is a via between `M1` and `M2`, `VR12` is a via between `R1` and `M2` (these are two separate etch steps).

To express this structure, one can use

```
PhysLayer M1
Conductor
Thickness 0.2

PhysLayer I1A
ViaCut VM12
Thickness 0.2

PhysLayer R1
Thickness 0.1
Rsh 2.0

PhysLayer i1B
ViaCut VM12|VR12
Thickness 0.2

PhysLayer VM12
Via M1 M2

PhysLayer VR12
Via R1 M2

PhysLayer M2
Conductor
Thickness 0.2
```

In this case, the `VM12` and `VR12` layers have no thickness, so they are abstract, but still establish connectivity for netlisting and LVS. The physical structure in 3D is actually established by the `I1A` and `I1B` layers, which must have nonzero Thickness. These are used when 3D extraction (cross section or L/C extraction) is being performed.

Dielectric

This keyword is intended to specify an explicit capacitor dielectric, which is different from a `Via` layer. A layer can not have both keywords. This is primarily to support the capacitance extraction interface. A `Dielectric` layer is assumed to be clear-field, unlike `Via` layers, though the `DarkField` keyword can also be applied. Also unlike `Via` layers, `Dielectric` layers are not assumed to be planarizing by default.

Contact *layer* [*expression*]

This keyword specifies that the present layer may be in contact with *layer*, which has the Conductor attribute, and is to be grouped accordingly in the wire net extraction. The *expression* (which is a layer expression), if given, must be true in the overlap region between the object and the objects on *layer* for contact to be established.

The purpose is to account for a contact metalization which is applied over the normal wiring layers, which may itself be used for making connections occasionally. The Contact keyword implies Conductor. The Contact keyword should be given in the layer block of the contact metal layer. It is not necessary (or desirable) to include a reciprocal Contact specification in the referenced layer's block.

DarkField

This keyword indicates that the layer polarity on the chip is the reverse of that shown on-screen.

This is usually the case for via layers, for example, which are rendered as small squares to indicate the contact location, which is actually a hole in an insulating layer. At present, the only command that uses this keyword is the **Cross Section** command in the **View Menu**. Layers with the keyword applied will be shown as on-chip in the cross sectional view. This keyword is implicitly assigned by both Via and GroundPlaneClear.

The keyword has a secondary effect if used in conjunction with the GroundPlane (or the equivalent GroundPlaneDark) keyword. The combination is equivalent to GroundPlaneClear.

A.6.5 Technology File Layer Block Keywords: Physical Properties

The following keywords can appear only in physical layer fields, and they mostly specify physical material properties, or electrical parameters, used in various ways by the extraction system.

Many of these parameters are redundant or incompatible with each other. Warning messages may be issued when incompatibilities are detected, however unused information is usually simply ignored and does no real harm. In particular, there are two basic groups, those keywords that apply to conductors, and those that apply to insulators. Mixing these parameters on the same layer will likely generate a warning.

All of these settings can be entered with the **Edit Tech Params** command in the **Attributes Menu** and then written to disk with the **Save Tech** command in the **Attributes Menu**, or be entered with a text editor directly into the technology file.

Planarize [y|n]
: This specifies whether or not a layer is planarizing. This is used by the three-dimensional layer sequence generator when creating layer sequences for the capacitance extraction interface. The `Planarize` keyword can be applied to prevent planarization of layers that are planarized by default, or to force planarizing of layers that don't normally have this property. See the description of the sequence generator in 12.8 for a description of planarization, and which layers are planarized by default.

Thickness *thickness*
: This keyword supplies the film thickness of the corresponding deposited film. The *thickness* is given in microns. This can be applied to any physical layer.

FH_nhinc *nhinc*
: This keyword applies to the *FastHenry* interface, and may be applied to conducting layers. This specifies the `nhinc` parameter to horizontal segments (parallel to the substrate) with thinkness equal to the layer thickness (as given with the Thickness keyword). This is the number of filaments contained in the segment, which can account for skin or penetration depth of conductors on the layer, in the vertical direction. The value given must be an integer 1 or larger. See the *FastHenry* documentation for more information about the `nhinc` parameter.

FH_rh *rh*
: This keyword applies to the *FastHenry* interface, and may be applied to conducting layers. This specifies the `rh` parameter, which is the ratio of heights between adjacent filaments. The default ratio is 2.0. Note that this applies only when the number of filaments is larger than one. See the *FastHenry* documentation for more information about the `rh` parameter.

At most one of the following two keywords (Rho and Sigma) should be used.

Rho *resistivity*

This keyword supplies the resistivity, in MKS units (ohm-meters), of the corresponding conducting film. If Rsh (below) and Thickness are both given, then the resistivity is already available and this keyword is redundant. Supplying this keyword overrides the Rsh*Thickness value for the resistivity, when resistivity is used explicitly in the extraction system (in the inductance/resistance extraction interface).

Sigma *conductivity*

This keyword supplies the conductivity, in MKS units, of the corresponding conducting film. This is converted to resistivity (1.0/*conductivity*) internally, i.e., it is equivalent to giving Rho.

Rsh *ohms_per_square*

The single parameter is a floating point number giving the ohms per square value of the conducting material. This is used in computation of the resistance value of resistor devices. If Rho or Sigma is given, and also Thickness, then the sheet resistance is already available and this keyword is redundant. Supplying this keyword overrides the Rho/Thickness value for sheet resistance.

Tau *tau*

This is the Drude relaxation time for resistive layers, as accepted by current releases of FastHenry. This enables extraction of the parasitic inductance of resistors, which can become appreciable for some materials at low temperature. This parameter is used only when creating FastHenry input.

EpsRel *diel_constant*

This keyword supplies the relative dielectric constant of insulating layers.

Capacitance *units_per_sq_micron* [*units_per_micron*]

This enables computation of the capacitance of a conductor group on the present conducting layer. The first parameter is a floating point number giving capacitance per square micron. The optional second parameter (default 0) is the edge capacitance, per micron. The extracted capacitance is the conductor group area multiplied by the first parameter, plus the conductor group perimeter length multiplied by the second parameter, if given. The capacitance for each wire net is computed during extraction, and will be printed (if enabled) in the physical netlist output file.

The keyword "Cap" is accepted as an alias for "Capacitance".

Lambda *pene_depth*

This keyword specifies the London penetration depth of superconducting conductors, in microns. When Lambda is given, Rho/Sigma (if given) represents the conductivity due to unpaired electrons from the two-fluid model.

Tline *grnd_plane_layer* [*diel_thick diel_const*]

This keyword will enable use of a microstrip model which computes transmission line parameters. A microstripline geometry is assumed, with an object on the present layer forming a strip over an infinite ground plane layer, separated by a homogeneous dielectric of constant thickness. No account is taken of "real" geometry, except for the dimensions of the strip on the present layer.

The first argument is the name of a layer assumed as the ground plane. Both the present layer and the ground plane layer must be conductors and have Thickness and, if superconductors, Lambda defined. Non-superconductors are treated as perfect conductors.

The second argument is the assumed height, in microns, of the intervening dielectric. The third argument is the relative dielectric constant. If either or both of these arguments is missing or given as "0" (zero), then *Xic* will search for a layer with the Via keyword set that contains the present and the ground plane layers, and obtain the missing values from that layer.

Antenna *float_value*

This keyword applies to the **!antenna** command, and is meaningful on conducting layers. The *float_value* is a threshold antenna ratio, as explained for the **!antenna** command. The value is effectively passed to that command as a default for the layer.

A.6.6 Technology File Layer Block Keywords: Design Rules

The layer block may contain design rule specifications, which begin with a keyword. These keywords can appear only in physical layer blocks. See the description of the design rules in 15.3 for information regarding these keywords. The rules can be programmed from within *Xic* with the **Design Rule Editor**. These keywords are not recognized in the *XicII* and *Xiv* feature sets.

A.7 Technology File Standard Via Definitions

Xic provides support for OpenAccess/Virtuoso-style standard vias (see 5.8). These definitions are imported from a Virtuoso ASCII technology file when the `ReadCdsTech` is used to source a Cadence technology database, if any ¡tt¿standardViaDefs¡/tt¿ nodes exist. They will be written to and read from the *Xic* technology file using syntax described in this section.

Standard via definitions will be written following the derived layers when a new technology file is being created. This is the recommended location when hand editing a technology file. The definitions are required to follow the layer definition blocks of any layers used, but otherwise location in the technology file is flexible.

The syntax for a standard via definition is as follows.

`StandardVia` *viaName layer1 layer2 cutLayer cutWidth cutHeight cutRows cutCols* \
cutSpace_x cutSpace_y layer1Enc_x layer1Enc_y layer2Enc_x layer2Enc_y \
layer1Off_x layer1Off_y layer2Off_x layer2Off_y originOff_x originOff_y \
[*implant1 imp1Enc_x imp1Enc_y* [*implant2 imp2Enc_x imp2Enc_y*]]

The terms correspond to the options shown in the **Via Creation** panel from the **Edit Menu**, and their effects are described in that section. The definition must appear on a single logical line, but backslash line continuation (as shown) can be employed to break the line for improved human readability.

The line must begin with the `StandardVia` keyword. The remaining tokens are as follows. All of the numerical values can be altered by the user before placement, the values provided in the definition are the initial defaults. The layer names, however, can not be changed subsequently. All dimensions are in microns.

viaName

This is a unique name for the standard via, and can be any text word that can be used as a cell name. One convention is to use the layer names of the twe conductors, top conductor first, separated by an underscore (e.g., "`M2_M1`").

layer1 layer2 cutlayer

The three tokens that follow are the names of the bottom conductor, the top conductor, and the via layer, in that order. These layers must have been already defined in the technology file.

cutWidth cutHeight
These are floating-point numbers giving the size of the cut in microns. The cut is always rectangular.

cutRows cutCols
These are integers not less than 1, which indicate that the cut should be arrayed according to the numbers of rows and colums given. These are both usually 1 in a standard via definition, representing a minimum via. The user can array the cuts when necessary from the **Via Creation** panel.

cutSpace_x cutSpace_y
These apply when the cut is arrayed, and provide the edge-to-edge space between cuts in the x and y direction. This is usually a minimum value given by a design rule.

layer1Enc_x layer1Enc_y layer2Enc_x layer2Enc_y
These four dimensions provide the enclosure distance for the bottom (layer1) and top conductor rectangles relative to the cut. The enclosure is the distance that the metal rectangle extends outside of the cut area. This is usually a minimum value given by a design rule.

layer1Off_x layer1Off_y layer2Off_x layer2Off_y
These four dimensions provide offsets for the center of the two conductor rectangles relative to the center of the cut. These values are unlikely to be other than zero.

originOff_x originOff_y
These coordinates provide the origin of the sub-master cell relative to the center of the cut array. It is the location that corresponds to the mouse pointer when a new via instance is placed. These are unlikely to be other than zero.

All of the terms mentioned thus far are required. The remaining terms are optional.

implant1 imp1Enc_x imp1Enc_y
This is the name of a layer followed by two dimensions. If found, an additional rectangle of *implant1* is centered over the *layer1* (bottom conductor) rectangle. The enclosure values specify the distance the implant extends outside of the conductor, in the x and y directions.

implant2 imp2Enc_x imp2Enc_y
These may follow an *implant1* group only. This is the name of a layer followed by two dimensions. If found, an additional rectangle of *implant2¿* is centered over the *layer2* (top conductor) rectangle. The enclosure values specify the distance the implant extends outside of the conductor, in the x and y directions.

Standard via definitions successfully read from the technology file will be saved internally, and the definitions can be accessed from the **Via Creation** panel. The panel allows the default values to be overridden, and new vias to be created and placed. If no stantard via definitions were successfully read, the panel is unavailable and the **Create Via** button in the **Edit Menu** is grayed.

A.8 Technology File Attributes

The keywords described below appear (by convention) after the layer specifications, and control various global attributes of *Xic*. These are broken down into categories, which are presented in the order in which they will be written to a new technology file created by *Xic*. Actual order in the file is unimportant. The categories are:

Grid Presentation
Display options for the grid, which can be adjusted from within *Xic* in the **Style** page of the **Grid Setup** panel.

Misc. Presentation
Other general display attributes that correspond to the entries in the **Main Window** sub-menu of the **Attributes Menu**.

Attribute Colors
Colors used for background, highlighting, etc.

Grid and Edge Snapping
Parameters for grid spacing and edge snapping, which can be adjusted from within *Xic* in the **Snapping** page of the **Grid Setup** panel.

Function Key Assignments
Command mapping to keyboard function keys.

Grid Registers
Saved grid register contents.

Layer Palette Registers
Saved palette register contents.

Font Assignments
Fonts used by the graphical user interface.

Keyword Variables
Variable initialization as keywords.

Keywords listed in the first three categories (**Grid Presentation**, **Misc. Presentation**, and **Attribute Colors** can also appear in print driver blocks, in which case they are in effect when printing with that driver.

In the syntax descriptions, the italicized quantities represent data the needs to be provided. The "`y|n`" symbol implies that one of '`y`' or '`n`' should follow the keyword. Actually, '`0`' (zero), or any word that begins with the letters or sequence (case insensitive) '`n`', '`f`', '`of`' is taken as a false value. Anything else, including no following text, is taken as true ('`y`' is always redundant).

A.8.1 Grid Presentation

These keywords define the appearance of the axes and grid shown in the drawing windows on program startup. Within *Xic*, the presentation can be modified from the **Style** page of the **Grid Setup** panels associated with the drawing windows. The parameters given in the technology file apply to the main window, which are inherited by sub-windows when created. The parameters can subsequently be changed with the panel on a per-window basis.

For the main drawing window, the **Main Window** sub-menu of the **Attributes Menu** provides the **Set Grid** button, which brings up the **Grid Setup** panel. Sub-windows have the **Grid Setup** panel available from the **Attributes** menu in the sub-window. Pressing **Ctrl-g** while a drawing window has focus will also bring up the panel.

The keywords described in this section can also appear within print driver blocks. If they appear in a print driver block, the attribute will apply on-screen when that driver is active in printing mode, and in the printer output.

`Axes [Plain | Mark | None]`
This determines the presentation style for the axes in physical mode. The default is `Mark`, where the origin is marked with a small box. If `Plain` is given, the axes are simple lines. If `None` is given, the axes will not be drawn.

`ShowGrid` [y|n]
This determines whether or not the grid will be shown by default, and applies to both physical and electrical modes.
Default: `y`

`ElecShowGrid` [y|n]
`PhysShowGrid` [y|n]
These keywords allow the grid display to be set independently for the two modes. The last `ShowGrid` directive will have precedence for a given mode.

`GridOnBottom` [y|n]
This keyword determines whether the grid is shown on top of or below the rendered objects.
Default: `y`

`ElecGridOnBottom` [y|n]
`PhysGridOnBottom` [y|n]
These keywords allow the grid to be displayed above or below the rendered objects independently for the two modes. The last `GridOnBottom` directive will have precedence for a given mode.

`GridStyle` *style* [*xsize*]
This sets the style of grid to use in both electrical and physical modes. The style is a decimal of hex (with "`0x`" prefix) integer whose binary pattern is used to replicate the grid lines. A value of 0 indicates a point grid, and -1 indicates solid grid lines. Other values are taken as a line pattern that is periodically reproduced. From the MSB, the pattern starts with the first set bit, and continues through the LSB.

If the *style* value is 0, for a "dots" grid, a second integer will be read if present. This value can be 0–6, and represents the number of pixels to light up around the central pixel in the four compass directions. The "dots" can appear as brighter dots or small crosses, as set by this integer. This integer is ignored if *style* is nonzero, and is taken as 0 if absent.
Default: 0xcc (hex)

`ElecGridStyle` *style*
`PhysGridStyle` *style*
These keywords allow the grid style to be set independently for electrical and physical modes. The last `GridStyle` directive has precedence for a given mode.

`CoarseGridMult` *num*
This can be set to an integer 1–50, and specifies that coarse grid lines will appear every *num* fine grid lines. With value 1, the grid will use the coarse grid color only. This applies in both electrical and physical modes.
Default: 5

`ElecCoarseGridMult` *num*
`PhysCoarseGridMult` *num*
These provide the coarse grid multiplier independently for the two modes. The last `CoarseGridMult` directive seen for a given mode has precedence.

A.8.2 Misc. Presentation

These keywords set initial values for a number of display attributes. These generally apply to all drawing windows, but the values can be reset on a per-window basis within *Xic*. For the main window, most have corresponding toggle buttons in the **Main Window** sub-menu of the **Attributes Menu**. In sub-windows, the buttons are located within the **Attributes** menu itself.

The keywords described in this section can also appear within print driver blocks. If they appear in a print driver block, the setting will apply on-screen when that driver is active in printing mode, and in the printer output.

`Expand` *num*
This keyword sets the initial expansion level for subcells, for both electrical and physical modes. If zero, no subcells are expanded. If -1, all subcells will be shown expanded. A positive integer indicates that subcells up to that depth will be shown expanded.
Default: 0

In *Xic*, the **Expand** pop-up controls expansion level, on a per-window basis. This panel is available from the **Expand** button in the main and sub-window **View** menus.

`ElecExpand` *num*
`PhysExpand` *num*
These forms allow the expansion level for electrical and physical modes to be set separately.

`DisplayAllText` *num*
This keyword sets whether label text is displayed or not, for both electrical and physical modes. If *num* is 0, labels will not be displayed. If 1 (actually, any number not 0 or 2), labels will be displayed in "legible" orientation. If 2, labels will be shown in true orientation, i.e., rotated and mirrored as placed and transformed along with the containing instance.
Default: `1`

The **Show Labels** and **Label True Orient** buttons in the **Main Window** sub-menu of the **Attributes Menu** and in the **Attributes** menu of sub-windows control these settings.

`ElecDisplayAllText` *num*
`PhysDisplayAllText` *num*
These forms allow the display of label text for electrical and physical modes to be set separately.

`ShowPhysProps` [y|n]
This keyword sets whether physical property strings are displayed in physical mode.
Default: `n`

The **Show Phys Properties** button in the **Main Window** sub-menu of the **Attributes Menu** and in the **Attributes** menu of sub-windows controls this setting.

`LabelAllInstances` *num*
This keyword sets whether unexpanded instances are labeled or not, for both electrical and physical modes. If *num* is 0, instances will not be labeled. If 1, instances will be labeled, with the label appearing either in horizontal or vertical orientation, whichever provides the best fit into the cell bounding box. If 2, the cell name is rotated and mirrored along with the cell.
Default: 1

The **Show Cell Names** and **Cell Name True Orient** buttons in the **Main Window** sub-menu of the **Attributes Menu** and in the **Attributes** menu of sub-windows control these settings.

`ElecLabelAllInstances` *num*
`PhysLabelAllInstances` *num*
These forms allow the display of unexpanded instance text for electrical and physical modes to be set separately.

`ShowContext` [y|n]
When given 'y', the context surrounding a subcell is shown during a sub-edit initiated with the **Push** command in the **Cell Menu**. This applies to both electrical and physical modes.
Default: `y`

The **Show Context in Push** button in the **Main Window** sub-menu of the **Attributes Menu** and in the **Attributes** menu of sub-windows controls this setting.

`ElecShowContext` *num*
`PhysShowContext` *num*
These forms allow the display of editing context for electrical and physical modes to be set separately.

`ShowTinyBB` [y|n]
If 'y' is given, tiny subcells will be represented by their bounding box. Otherwise, these subcells will not be shown. The size threshold is given by the CellThreshold variable, set with the **!set** command. This applies to both electrical and physical modes.
Default: `y`

The **Subthreshold Boxes** button in the **Main Window** sub-menu of the **Attributes Menu** and in the **Attributes** menu of sub-windows controls this setting.

`ElecShowTinyBB` *num*
`PhysShowTinyBB` *num*
These forms allow the tiny subcell rendering for electrical and physical modes to be set separately.

A.8.3 Attribute Colors

The following keywords set colors used on-screen and in printer output. All of these keywords take a *colorspec* string as the argument list. This is the name of a color or an RGB triple:

- The name of a color. The recognized names can be listed from the **Set Color** pop-up in the **Attributes** menu with the **Colors** button.
- Three space-separated numbers, each 0–255, representing the red, green, and blue intensity. E.g., "`196 240 235`".
- Other forms recognized by the `XParseColor` C library function, including "`#RRRRGGGGBBBB`" and "`rgb:RRRR/GGGG/BBBB`". Here, `R`, `G`, and `B` are single hexadecimal digits.

Following the general pattern for the technology file keywords, the keyword form without the "Phys" or "Elec" prefix sets the color for both modes. The mode-specific keywords set the color only for that mode.

A single internal data structure maintains all other attribute (non-layer) colors. All attribute colors can be set from the **Color Selection** panel provided by the **Set Color** button in the **Attributes Menu**. Attribute colors can also be changed with the **!setcolor** command. In Unix/Linux, colors can be initialized from a resource file (see A.10), as well as from the technology file.

When *Xic* starts, the colors are set to default values. Then, any colors found in a resource file are updated. Then, some of the colors may be modified in the technology file. Finally, the colors may be changed in a `.xicstart` file.

Below is the list of attribute colors, the defaults, and techfile keywords and aliases. The `SelectColor1/2` set the blinking highlighting used for selected objects. Setting both to the same color stops the blinking. The `MarkerColor` is used for electrical-mode terminal marks. The **Plot Mark** colors are used only for the plot point indicators, and match the colors defined for plots in *WRspice*.

The **Prompt Line Colors** apply tho the prompt line, status area, coordinate readout, and main window keys-pressed area. The `PromptBackgroundColor` controls the common background color, except when the prompt line is in editing mode. The other colors are self-explanatory, with the `PromptHighlightingColor` being the color used for hypertext entries (mostly for electrical mode).

The **Special GUI Colors** are miscellaneous colors used for highlighting and other purposes in the graphical user interface.

Variable	Use
`GUIcolorDel`	**Cell Hierarchy Digests**, **File Selection**, etc.
`GUIcolorNo`	**Empty Cells**, **Modified Cells**, **Set Cell Flags**
`GUIcolorYes`	**Empty Cells**, **Modified Cells**, **Set Cell Flags**
`GUIcolorHl1`	**Script Debugger**, **Design Rule Editor**, **Property Editor**
`GUIcolorHl2`	**Modified Cells**, **Property Editor**, **Cell Property Editor**
`GUIcolorHl3`	**Modified Cells**
`GUIcolorHl4`	**Design Rule Editor**, **Tech Parameter Editor**, **Property Editor**, **Cell Property Editor**
`GUIcolorDvBg`	Pictorial device menu background
`GUIcolorDvFg`	Pictorial device menu foreground
`GUIcolorDvHl`	Pictorial device menu highlight
`GUIcolorDvSl`	Pictorial device menu selection

The **Attribute Colors** listed in the first block in the table below can also be specified in printer driver blocks. In this case, the color will apply when that driver is selected in print mode, both on-screen and in the hard-copy output generated by the driver.

Keyword Alias	Default
Attribute Colors	
GhostColor	white
ElecGhostColor	GhostColor
PhysGhostColor	GhostColor
HighlightingColor Highlighting	white
ElecHighlightingColor ElecHighlighting	HighlightingColor
PhysHighlightingColor PhysHighlighting	HighlightingColor
SelectColor1	white
ElecSelectColor1	SelectColor1
PhysSelectColor1	SelectColor1
SelectColor2	pink
ElecSelectColor2	SelectColor2
PhysSelectColor2	SelectColor2
MarkerColor	yellow
ElecMarkerColor	MarkerColor
PhysMarkerColor	MarkerColor
InstanceBBColor InstanceBB InstanceBox	turquoise
ElecInstanceBBColor ElecInstanceBB ElecInstanceBox	InstanceBBColor
PhysInstanceBBColor PhysInstanceBB PhysInstanceBox	InstanceBBColor
InstanceNameColor InstanceName	pink
ElecInstanceNameColor ElecInstanceName	InstanceNameColor
PhysInstanceNameColor PhysInstanceName	InstanceNameColor
InstanceSizeColor InstanceSize	salmon
CoarseGridColor CoarseGrid	sky blue
ElecCoarseGridColor ElecCoarseGrid	CoarseGridColor
PhysCoarseGridColor PhysCoarseGrid	CoarseGridColor
FineGridColor FineGrid	royal blue
ElecFineGridColor ElecFineGrid	FineGridColor
PhysFineGridColor PhysFineGrid	FineGridColor

Keyword Alias	Default
Prompt Line Colors	
PromptTextColor PromptText	sienna
PromptEditTextColor PromptEditText	black
PromptHighlightColor PromptHighlight	red
PromptCursorColor PromptCursor	blue
PromptBackgroundColor PromptBackground	gray92
PromptEditBackgColor PromptEditBackg PromptEditBackground	gray96
PromptEditFocusBackgColor PromptEditFocusBackg PromptEditFocusBackground	gray100
Plot Mark Colors	
Color2	red
Color3	lime green
Color4	blue
Color5	orange
Color6	magenta
Color7	turquoise
Color8	sienna
Color9	gray
Color10	hot pink
Color11	slate blue
Color12	spring green
Color13	cadet blue
Color14	pink
Color15	indian red
Color16	chartreuse
Color17	khaki
Color18	dark salmon
Color19	rosy brown
Special GUI Colors	
GUIcolorSel	#e1e1ff
GUIcolorNo	red
GUIcolorYes	green3
GUIcolorHl1	red
GUIcolorHl2	darkblue
GUIcolorHl3	darkviolet
GUIcolorHl4	sienna
GUIcolorDvBg	gray90
GUIcolorDvFg	black
GUIcolorDvHl	blue
GUIcolorDvSl	gray80

A.8.4 Grid and Edge Snapping

These keywords define the grid and edge snapping parameters. These can be reset from within *Xic* from the **Snapping** page of the **Grid Setup** panels associated with the drawing windows. The parameters given in the technology file apply to the main window, which are inherited by sub-windows when created. The parameters can subsequently be changed with the panel on a per-window basis.

For the main drawing window, the **Main Window** sub-menu of the **Attributes Menu** provides the **Set Grid** button, which brings up the **Grid Setup** panel. Sub-windows have the **Grid Setup** panel available from the **Attributes** menu in the sub-window. Pressing **Ctrl-g** while a drawing window has focus will also bring up the panel.

`MfgGrid` *delta*

If set nonzero, the actual `SnapGridSpacing` used will be constrained to be a multiple of this value. This applies in physical mode only.

This can be considered to be the "pixel" size of the mask. The SpotSize variable is related, see this topic in E.11 for more information.

`SnapGridSpacing` *spacing*

The *spacing* is a floating point number which represents the spacing, in microns, between snap points. This applies to physical mode only.
Default: 1.0 microns

The electrical grid is set to spacing value 1.0 with unit snap per grid on program startup, which can't be changed from the technology file. The electrical grid can be changed within *Xic* from the **Grid Parameters** pop-up, in the unusual circumstance that non-default values are needed.

`SnapPerGrid` *num*
`GridPerSnap` *num*

At most one of these keywords should be given. The *num* is an integer 1–10. These apply to physical mode only.

If `SnapPerGrid` is given, then the fine grid lines will be spaced *num*`*SnapGridSpacing` apart. If `GridPerSnap` is given, fine grid lines will appear at `SnapGridSpacing/`*num* intervals.
Default: 1

`EdgeSnapping` [`none`|`some`|`all`] [`+`|`-off_grid`] [`+`|`-non_manh`] [`+`|`-edge_of_wire`] [`+`|`-path_of_wire`]

This keyword sets the initial state of the controls of the **Edge Snapping** group in the **Snapping** page of the main window **Grid Setup** panel. All fields are optional, with the effective default being

```
EdgeSnapping some -off_grid -non_manh +edge_of_wires -path_of_wires
```

Only the first letter of the keywords is needed, and recognition is case-insensitive.

The first word specifies when edge snapping is enabled:

`none`	not enabled
`some`	enabled in some commands (the default)
`all`	always enabled

The remaining items are flags that must start with a `+` or `-` character. The `+` turns the option on, the `-` turns the option off. These have obvious correspondence to the check boxes in the **Edge Snapping** control group in the **Grid Setup** panel, and set the initial state of the check boxes for the main window.

RulerEdgeSnapping [none|some|all] [+|-off_grid] [+|-non_manh] [+|-edge_of_wire] [+|-path_of_wire]
This keyword uses the same syntax as the EdgeSnapping keyword, and species the initial edge snapping mode when the **Rulers** command in the **View Menu** is in effect. This command has its own settings, with the default being that all flags are enabled.

RulerSnapToGrid [y|n]
This boolean keyword specifies the initial state of grid snapping in the **Rulers** command in the **View Menu** In the command, the mode can be toggled by presing the period ('.') key. By default, grid snapping is asserted.

A.8.5 Function Key Assignments

It is possible to map the keyboard function keys to *Xic* operations. The function key assignments are sensitive to the **Shift**, **Control**, and **Alt** keys. This means that a function key (**F1** - **F12**) press can have different effects depending on the state of these keys.

F*N*Key *text*
The *N* is an integer in the range 1–12, to correspond to the **F1** – **F12** function keys found on most keyboards.

The *text* has the form

[<*tok*>] *cmd* [<tok> *cmd*] ...

Each *tok* is a combination of the letters s, c, and a. The presence of the letters indicates that **Shift**, **Control** and **Alt** are pressed, respectively. The *tok* is surrounded by angle brackets.

Examples:

<s>
The **Shift** key is pressed, **Control** and **Alt** are not pressed.

<ca>
The **Control** and **Alt** keys are pressed, the **Shift** key is not pressed.

These tokens are followed by a *cmd*, which is a command. If the command starts with '!', the remainder is treated as a "bang" command (see 19). Otherwise, the text is the five-character (or fewer) command keyword associated with GUI command buttons. If the *cmd* contains white space, it must be quoted.

The command keywords are displayed in the pop-up "tooltip" which appears when the mouse pointer is positioned over a command button, after a short delay. This is the internal name for the command, which is generally a short mnemonic of five characters or fewer. The keywords are also generally provided in the help system topic describing the command. In the **User Menu**, for user scripts, the name which appears on the menu button is the appropriate name to use.

The first *tok* is generally absent, and the *cmd* applies to the function key with no modifiers pressed.

Example:

```
F1Key box <s> "!exec /path/to/myscript.scr" <c> !!Clear(0) <sca> polyg
```

The terms are:

F1Key
We're setting the **F1** key in this example.

`box`
This indicates that when **F1** is pressed without pressing a modifier key, the "`box`" command from the side menu wil be started.

`<s> "!exec /path/to/myscript.scr"`
This specifies that when **Shift-F1** is pressed, the script in the `myscript.scr` file will be executed, using the **!exec** bang command. Since the command contains a space character, it is quoted.

`<c> !!Clear(0)`
This specifies that when **Control-F1** is pressed, the `Clear()` script function is called with argument 0. This will clear the database. Note that the single/double exclamation point syntax is the same as is accepted on the command line.

`<sca> polyg`
When **Shift**, **Control**, and **Alt** are pressed along with **F1**, the "`polyg`" polygon creation command from the side menu is started.

The menu containing the named button must be active (not grayed) for the function key to have effect. The mappings are completely defined by the user — there are no defaults. Pressing an unmapped function key has no effect on *Xic*. Be aware that the window manager in use, and the GTK toolkit, may map functions keys, and this may have higher priority than the mapping assigned here. The use of the **Alt** key is generally not a good idea, as it is commonly assigned for other purposes. Sometimes, an assignment will simply be ignored for some reason. For example, on one system **Control-F1** is never returned, but **Control** works fine with other function keys.

A.8.6 Grid Registers

The grid registers from the **Grid Setup** pop-up are saved in the technology file if they contain a non-default grid.

`ElecGridReg`*N spec*
`PhysGridReg`*N spec*
The *N* is an integer value in the range 1–7. Each register index can store both a physical and electrical grid specification. The specifications define the contents of the grid registers, available in the **Grid Setup** panel and elsewhere.

The *spec* string has the form:

snapspace snapval linestyle [*xsize*] [`-a` *axes*] [`-d` *dsp*] [`-t` *ontop*] [`-m` *cmult*]

The first three tokens are mandatory, and must appear in the order shown.

snapspace (real number)
The spacing between snap points, in microns.

snapval (integer, -10 through 10 excluding 0)
If the value is positive, it sets the number of snap points per fine grid line. For example, a value of 3 would indicate that a fine grid line is drawn at every third snap point. If negative, this sets the number of fine grid lines per snap interval. In this case, a value of three indicates that fine grid lines appear at snap points and at the 1/3 and 2/3 proportional distances within the snap interval.

linestyle (integer)
This is the line style code. The value is 0 for a dot grid, otherwise the bit pattern represents the line dashes, as for the `GridStyle` keyword.

The remaining tokens are optional, and can follow the first three in any order.

xsize (integer 0–6)
If the *linestyle* code is 0 (for a dot grid), then a fourth number can appear. This is an integer 0–6 which indicates the number of pixels in the four orthogonal directions to extend the dot into a cross.

`-a` *axes* (integer 0–2)
This sets the axes presentation mode in physical mode. If 0, axes are shown and the origin decorated. If 1, plain axes are shown, and if 2, axes aren't shown.

`-d` *dsp* (boolean)
This sets whether the grid is displayed or not. The *dsp* token can be about any alphanumeric token that by convention indicates true or false.

`-t` *ontop* (boolean)
This sets whether the grid is displayed after all geometry ("on top") or before geometry. The *ontop* token can be about any alphanumeric token that by convention indicates true or false.

`-m` *cmult* (integer 1–50)
This sets the number of fine grid lines per coarse grid line.

For backward compatibility, "`GridReg`" is accepted as "`PhysGridReg`".

A.8.7 Layer Palette Registers

The palette registers from the **Layer Palette** are saved to and assigned from the technology file.

`ElecLayerPalette`*N layer_list*
`PhysLayerPalette`*N layer_list*
The *N* is an integer value in the range 1–7 that specifies a register number. The *layer_list* is a list of layer names separated by white space. There are separate entries for electrical and physical mode for each register number. The list provides the layer names and ordering of the layers in the "user" part of the layer palette.

A.8.8 Font Assignments

The keywords described below set the fonts used in various places in *Xic*. These correspond to the fonts settable from the **Font Selection** pop-up from the **Set Font** button in the **Attributes Menu**.

Since the font string format varies between the operating systems and graphical interfaces supported by *Xic*, provision is made for separate font specifications for each supported variation, thus making the technology file more portable between different versions of *Xic*.

There are six fonts that may be set, and four sets of corresponding keywords, specific to different systems. The four sets correspond to a suffix character added to the font keyword.

`Font1 --- Font6` *name_of_font*
These keywords will be read and (if possible) applied by any version of *Xic*. Although there is an attempt at portability, the *name_of_font* should apply to the release of *Xic* in use. A mismatch will not cause errors, but the font may not be as expected, or a default may be used. These keywords are mostly for backwards compatibility, and are never written to a new technology file created with the **Save Tech** button in the **Attributes Menu**. Rather, the system-specific keywords below will be written.

`Font1P --- Font6P` *name_of_font*
These fonts apply to the releases that use the GTK-2 (Pango) font system. At the 3.3 release level, all *XicTools* programs use this graphical toolkit, and will use these keysords.

`Font1X --- Font6X` *name_of_font*
These keywords apply to non-current releases (FreeBSD7, Linux2, OS X) that use the GTK-1 X-windows font system. The *name_of_font* is the X Logical Font Descriptor for a font available on the user's system, or an alias. These font specifications are ignored in GTK-2 (all current) releases.

`Font1W --- Font6W` *name_of_font*
These keywords apply only to the non-current Microsoft Windows release, which used native Win32 for the graphical interface. There is really no syntactical difference between these and Pango (`P`) specifications, and (current) GTK-2 releases will accept (but not write) these.

If a font is specified more than once in the technology file, such as with duplicate or equivalent keywords, the last specification read will take precedence.

When a new technology file is written, only the keywords for non-default fonts in use will actually be written in the file.

The index number of the keyword indicates the following fonts:

`1` (Fixed Pitch Text Window Font)
This sets the font used in pop-up multi-line text windows other than the text editor/file browser, such as the **Files Listing** and **Cells Listing**, where the names are formatted into columns.
Defaults:
Unix/Linux: `Monospace 9`
Windows: `Lucida Console 9`

`2` (Proportional Text Window Font)
This sets the font used in pop-up multi-line text windows other than the text editor/file browser, where text is not formatted, such as the **Info** and error message pop-ups.
Defaults:
Unix/Linux: `Sans 9`
Windows: `Sans 9`

`3` (Fixed Pitch Drawing Window Font)
This is the font used in the coordinate readout, the status line, layer table, and the prompt line. It is not the font used to render label text in the drawing windows, which is a vector font generated by other means.
Defaults:
Unix/Linux: `Monospace 9`
Windows: `Lucida Console 9`

`4` (Text Editor Font)
This is the font used in the **Text Editor** and **File Browser** pop-ups.
Defaults:
Unix/Linux: `Monospace 9`
Windows: `Lucida Console 9`

`5` (HTML Viewer Proportional Font)
This is the base font used for proportional text in the HTML viewer (help windows). If set, this will override the font set in the `.mozyrc` file, if any.
Defaults:

Unix/Linux: `Sans 9`
Windows: `Sans 9`

`6` (HTML Viewer Fixed Pitch Font)
This is the base fixed-pitch font used by the HTML viewer. If set, this will override the font set in the `.mozyrc` file, if any.
Defaults:
Unix/Linux: `Monospace 9`
Windows: `Lucida Console 9`

The platform-specific font keywords were added in release 3.1.6. Older technology files will use only the `Font1 --- Font6` keywords. It may be be best to comment these out when importing a technology file developed for another platform, or to modify the `Font` keywords to the appropriate flavor with a text editor.

Fonts can be set within *Xic* with the **Set Font** command in the **Attributes Menu**.

A.8.9 Variable Setting as Keywords

In addition to the keywords described in the previous sections, most of the variables (see E) that are known to *Xic* can be set as keywords. These variables control various aspects of *Xic*, including the states of most of the controls in the various pop-up panels. When a technology file is written, variables that participate in this protocol and are set will contribute a corresponding line to the attributes section of the new technology file.

Most variables participate in the protocol. A few do not, for one reason or another, and it is unlikely that these will be missed. The **!attrvars** command will produce a list of the variables that participate, the user can check this if necessary.

When a new technology file is being written, variables that are set will generate content. There is no "default", and the options in the **Write Tech File** panel that alter the treatment of "default definitions" have no effect on these lines.

The same variables can also be set with the `!set` lines. If a variable is set multiple times by any means, the last one seen will have precedence. The variables that participate in the protocol but are set with the `!set` line will not be remembered as having been set. When a technology file is written, the remembered variables are given `!set` lines in the new file. This is not necessary for variables that participate in the protocol.

Variables are logically divided into classes. Boolean variables are switches that are either set (usually to an empty string) or not set. Other variables we refer to as "string" variables. They are set to an arbitrary text string, when set at all.

In the technology file, booleans take the form

VariableName [`y`|`n`]

which is the same syntax as for boolean keywords. The "`y`|`n`" symbol implies that one of '`y`' or '`n`' should follow the keyword. Actually, '`0`' (zero), or any word that begins with the letters or sequence (case insensitive) '`n`', '`f`', '`of`' is taken as a false value. Anything else, including no following text, is taken as true ('`y`' is always redundant). If the second token indicates affirmative, then the variable will be set. If the second token is negative, no action is taken.

String variables take the form

VariableName arbitrary text

where the variable will be assigned the *arbitrary text*, with leading and training white space stripped.

The **!attrvars** command lists the variables that are boolean and string separately, so the user can check this list if unsure of the variable type.

The *VariableName* in this context is recognized as a known variable name without case sensitivity. In every other context, variable names are case-sensitive. Since this syntax applies only to internal variable names, there is no conflict as there are no such variables that differ only in case.

A.9 Hardcopy Driver Parameters

By default, all hardcopy drivers available within the program are made available to the user through the **Format** menu in the **Print Control Panel**. Drivers can be disabled, so they don't appear in the **Format** menu, by adding the "`off`" keyword to the "`HardCopyDevice`" line, which begins the block of lines describing the driver defaults. The driver blocks are found near the end of the technology file, and are written in their entirety when the **Save Tech** command is used to generate a technology file. It is not an error for a driver block to be absent; internal defaults will be used.

The following keyword(s) may be used outside of the driver blocks to set the default print driver.

`DefaultDriver` *driver_name*
This keyword sets the default print driver to use in both electrical and physical modes. When the **Print Control Panel** initially appears, the **Format** menu will have this driver selected. The *driver_name* is one of the driver names as listed in the `HardCopyDevice` keyword description below. The keyword `AltDriver` is recognized as a synonym for this keyword.

`ElecDefaultDriver` *driver_name*
Similar to `DefaultDriver`, but sets the default to use in electrical mode only. The keyword `AltElecDriver` is a synonym.

`PhysDefaultDriver` *driver_name*
Similar to `DefaultDriver`, but sets the default to use in physical mode only. The keyword `AltPhysDriver` is a synonym.

A driver block begins with a `HardCopyDevice` line naming the driver, and ends with the next `HardCopyDevice` line or end of file. In addition to the `HardCopy...` keywords that specify driver defaults, any of the keywords described in the **Presentation Attributes** and **Attribute Colors** categories of the **Technology File Attributes** section A.8 can be used. The attribute or color will then apply while in print mode and the driver is selected, both on-screen and in the driver output. The keyword formats are exactly as described in these subsections. If not given in a driver block, the driver will use the attribute or color values set in the main part of the technology file, or the program defaults if no value is specified.

Layer colors, fill, and visibility can be set on a per-layer basis for the driver, by including a "mini-layer block". This is a truncated version of the layer blocks described in **Technology File Layer Blocks**, section A.6. The only keywords which are accepted in a mini-layer block are `RGB` (to set the color), `Filled` (to set the fill pattern or outline style, and `Invisible` (to set visibility). However, there are two additional special keywords that may be included in specific drivers:

`HPGLfilled` *filltype* [*option1 option2*]

This keyword is recognized and used only by the HP-GL hard copy driver ("hpgl_line_draw_color"), and is used to specify a fill pattern for the layer (electrical or physical). The parameters are those appropriate for the `FT` HPGL directive, as documented in

> "The HP-GL2 and HP RTL Reference Guide: A Handbook for Program Developers"

from Hewlett-Packard, (ISBN 0-201-63325-6) pages 127-129. This is summarized below:

filltype	**description**	**option1**	**option2**
1	solid, bidirectional	ignored	ignored
2	solid, unidirectional	ignored	ignored
3	hatched, parallel lines	line spacing	line angle
4	crosshatched	line spacing	line angle
10	shadings	shading level	ignored
11	not supported	ignored	ignored

There are 1016 dots per inch and angles are in degrees. Shading level is 0–100. If the `HPGLfilled` keyword is supplied for a layer and the *filltype* and options (if given) are valid, that fill will be used with the layer in HPGL output. There is presently no way to assign the layer color.

This parameter must be added to the technology file with a text editor. The default is no fill. Note that the fill patterns set on the screen in hard copy mode are not used by the HP-GL driver.

`XfigFilled` *filltype*

This keyword is recognized and used only by the `xfig` hard copy driver ("xfig_line_draw_color"), and allows setting the fill patterns for the layer (electrical or physical). The *filltype* is an integer 1–56, which selects one of `xfig`'s internal fill patterns.

0	No fill
...	shades
20	Full saturation of the color
...	tints
40	White
41	30 degree left diagonal pattern
42	30 degree right diagonal pattern
43	30 degree crosshatch
44	45 degree left diagonal pattern
45	45 degree right diagonal pattern
46	45 degree crosshatch
47	Bricks
48	Circles
49	Horizontal lines
50	Vertical lines
51	Crosshatch
52	Fish scales
53	Small fish scales
54	Octagons
55	Horizontal "tire treads"
56	Vertical "tire treads"

Values 1 to 19 are "shades" of the color, from darker to lighter, a shade is defined as the color mixed with black. Values from 21 to 39 are "tints" of the color from the color to white, a tint is defined as the color mixed with white. The `XfigFilled` parameter must be added to the technology file with a text editor. The default is no fill. Note that the fill patterns set on the screen in hard copy mode are not used by the xfig driver.

As for regular layer blocks, a mini-layer block starts with a `PhysLayer` or `ElecLayer` keyword, or one of the aliases. The layer name given must be the name of a layer supplied in one of the regular layer blocks. A mini-layer block terminates when a new mini-layer block starts, or at the end of the driver block. The block order, and order with respect to other keywords, is arbitrary.

The other keywords of the driver block are described below.

`HardCopyDevice` *device_name* [off]

This line begins the driver block, and the keywords that follow apply to the *device_name* driver. The names are internally recognized strings:

```
hp_laser_pcl
hpgl_line_draw_color
postscript_bitmap
postscript_bitmap_encoded
postscript_bitmap_color
postscript_bitmap_color_encoded
postscript_line_draw
postscript_line_draw_color
windows_native
xfig_line_draw_color
image
```

If the "`off`" keyword is given ("`disable`" and "`n`" are synonyms), the driver is disabled, and will not appear in the **Format** menu of the **Print Control Panel**.

See the description of the **Print** button in the **File Menu** (8.6.2) for more information on these drivers.

`HardCopyLegend` *n*

This keyword sets the default status of the **Legend** button in the **Print Control Panel** when the driver is active. Values can be 0, 1, or 2:

0	**Legend** button is off
1	**Legend** button is on
2	**Legend** button is grayed and inactive

`HardCopyOrient` *n*

This keyword sets the default status of the **Portrait**, **Landscape**, and **Best Fit** buttons in the **Print Control Panel** while the driver is active. Values are 0–3:

bit 0 set	**Landscape** on, **Portrait** off
bit 0 unset	**Landscape** off, **Portrait** on
bit 1 set	**Best Fit** button on
bit 1 unset	**Best Fit** button off

`HardCopyCommand` *command_string*

Specifies the command to use to queue the plot. This will be shown in the command text box of the **Print Control Panel**. The characters "`%s`" will be replaced with the name of the temporary file, all other characters are passed verbatim. If "`%s`" does not appear in the string, the file name will be appended to the string, separated by a space character. This keyword is ignored under Microsoft Windows.

`HardCopyResol` *list_of_integers*
This sets the resolutions supported by the driver, in dots per inch.

`HardCopyDefResol` *integer*
This has meaning only to drivers that have selectable resolutions. The value following this keyword is a zero-based index into the list of resolutions as given with the `HardCopyResol` keyword, and indicates the default resolution which will be selected in the **Print Control Panel** for the driver.

Example:

```
HardCopyDevice postscript_line_draw
HardCopyResol 72 75 100 150 200 300 400
HardCopyDefResol 2
```

This will select 100 as the resolution for the `postscript_line_draw` driver when the **Print Control Panel** first appears. The resolution can be changed with the menu.

`HardCopyDefHeight` *float_format_number*
`HardCopyDefWidth` *float_format_number*
`HardCopyDefXoff` *float_format_number*
`HardCopyDefYoff` *float_format_number*
These set the default image size and location, and are in inches, unless followed by the letter 'c' which denotes centimeters. The `Yoff` number may be interpreted as a top or bottom margin, depending upon the driver. The dimensions are in all cases relative to the portrait orientation of the page. If the width or height is set to zero (but not both) the driver will assume auto-width or auto-hight mode, where the width or height is set to the minimum necessary to render the object.

`HardCopyMinHeight` *float_format_number*
`HardCopyMinWidth` *float_format_number*
`HardCopyMinXoff` *float_format_number*
`HardCopyMinYoff` *float_format_number*
These set the minimum acceptable values for the parameters.

`HardCopyMaxHeight` *float_format_number*
`HardCopyMaxWidth` *float_format_number*
`HardCopyMaxXoff` *float_format_number*
`HardCopyMaxYoff` *float_format_number*
These set the maximum acceptable values for the parameters.

A.10 Resource File

One can use the resource-setting capability of the X-Windows system to set attribute colors. This applies when running under the X-Windows system, which is presently true for all releases except those for Microsoft Windows. However, this is archaic and not really recommended.

One can create an X resource file for *Xic*. This is a file that should be created in the user's home directory, with a name that is the executable program name with the first letter capitalized, i.e., `Xic`. The file contains lines in the following form:

```
xic.HighlightingColor:  green
xic.MarkerColor:  blue
```

or generally

`xic.`*resourcename*`:` *colorspec*

The *resourcename* is a keyword from the list of attribute colors as listed in A.8.3. Note that the keyword must be used, not an alias. The aliases are recognized in the technology file and **!setcolor** command. The *colorspec* string is the name of a color or an RGB triple:

- The name of a color. The recognized names can be listed from the **Set Color** pop-up in the **Attributes** menu with the **Colors** button.
- Three space-separated numbers, each 0–255, representing the red, green, and blue intensity. E.g., "`196 240 235`".
- Other forms recognized by the `XParseColor` C library function, including "`#RRRRGGGGBBBB`" and "`rgb:RRRR/GGGG/BBBB`". Here, `R`, `G`, and `B` are single hexadecimal digits.

Appendix B

Design Data File Formats

This section describes the extensions to the CIF and GDSII formats used by *Xic*. The CGX file format, designed to be a more efficient replacement for GDSII and released into the public domain by Whiteley Research Inc., is described below as well. The extensions to CIF and GDSII are designed to accommodate the electrical information and certain properties. When strict conformance to the standard format is required, such as when exporting physical layouts to a mask vendor, the **Strip For Export** button in the **Export Control** panel should be used to strip out all extensions, leaving only the physical layout.

The GDSII (Stream) format is owned by Cadence, Inc., and is described in documentation available from Cadence (specifically the "Design Data Translators Reference," which is updated periodically), which may be available on the internet.

In *Xic*, cell names are not limited in length. The cell names can contain any characters valid in a file name with the exception of the semicolon (';'), which is reserved in the CIF-like syntax used for native cell files. For portability, it is recommended that cell names use only the GDSII allowed characters, which are the alpha-numerics plus '_', '$', and '?'. In older GDSII specifications (release 3), cell names were limited to 32 characters, so it may be wise to observe this limit in *Xic*.

Archive files created by *Xic* generally consist of two records, the first containing the physical information, and the second containing electrical information. If there is no electrical information, the second block is not written. Each block is an individual representation of the archive file type, i.e., they each parse as a complete "file". GDSII files written in this manner are generally portable to other CAD systems (format extensions appear in the electrical block only), as reading will terminate at the end of the physical block, and the following electrical block will be ignored. However, in specific instances where this proves not to be true, the **Strip For Export** button, which eliminates the electrical block, should be used.

The same comments apply to OASIS and CGX formats, however CGX is not known to be supported by other CAD vendors at this time.

CIF files use extensions unique to *Xic*, so will likely not be portable to other CAD systems unless **Strip For Export** is used. These extensions are described below.

B.1 GDSII Format and Extensions

The GDSII format provides a compact, binary representation of a design hierarchy. Though the standard format is intended for physical data, minor extensions are used by *Xic* to allow storage of the electrical information as well.

A GDSII file consists of a sequence of data blocks. The first four bytes of the block provide the block type and size. Integers and floating point numbers have defined representations in GDSII, so conversion is necessary from most machine representations. This section will describe the extensions only.

A GDSII file can be decomposed into an ASCII representation with the **Format Conversion** panel found in the **Convert Menu**. The resulting file prints out the characteristics of each block, plus the block offset within the file and messages indicating extensions and errors. A file in this format can be reconverted to GDSII. Generally, there is a one to one correspondence between items in the text representation and blocks in the GDSII file, thus one can learn much about the structure of the GDSII file by examining the text representation.

When the **Strip For Export** button in the **Export Control** panel is active, GDSII files produced from this panel adhere to the strict GDSII standard and contain only physical data. Otherwise the file produced will contain extensions. Such files are not guaranteed to be readable by other software (but they generally are).

The file with extensions contains two concatenated GDSII record sets. The first contains the physical data. The physical records are zero padded to the next 2Kb block boundary, at which point the electrical records begin. Beyond this arrangement the extensions are as follows:

1. The data size limitation on attribute strings is increased and the total size of attribute lists is unconstrained. Attribute strings can be up to 16Kb in length.
2. Attribute records can appear ahead of cell definitions, thus giving properties to cell definitions. This violates the standard GDSII record sequencing.

Both of these extensions are necessary to accommodate to properties found in the electrical design data, and are used in the electrical part of the file only.

B.1.1 Physical Mode Cell Properties

Certain features, such as parameterized cells, require cell properties in physical mode. Cell properties are also used to save the grid/snap values in the top-level cell, and can be added by the user to support other applications.

The GDSII format has no provision for storing properties of cell definitions. In electrical mode, *Xic* uses the format extension mentioned above. We can't use extensions in physical mode, since that would make the files non-portable, so we have to fake it.

In releases prior to 2.5.66, the cell properties were saved in a dummy label. This label was written on layer/datatype 0/0 at the origin, and was given the text string "`CELL PROPERTIES`". Physical cells with properties would have this label added in GDSII output. When reading in the GDSII file, the label would be stripped, and the properties from the label object would be applied to the containing cell.

However, when using direct conversions from the **Format Conversion** panel from the **Convert Menu**, the file would be converted as-is, so that if converting to *Xic* native cell files (for example), the converted cells would contain the "`CELL PROPERTIES`" labels and would **not** have the properties set.

Release 2.5.66 and later no longer create a "`CELL PROPERTIES`" label. Instead, a "SNAPNODE" record with a PLEX number `0xffffff` is created, at the origin on layer 0 with nodetype 0. This is an obscure data type that is more likely to be invisible in the GDSII database. Unlike a label, it should not be visible in most other readers.

The current release reader will still process the "`CELL PROPERTIES`" label if found, for backwards compatibility. In addition, it will also process these labels, and the SNAPNODE records, when doing direct conversions, so that the properties are assigned correctly in this case.

Neither construct is/was added to the output file if the StripForExport button or variable is active.

COMPATIBILITY WARNING
Xic releases prior to 2.5.66 will not process cell properties in GDSII files created with this release and subsequent. Physical mode cell properties are used by *Xic* to implement parameterized cells, to save the grid parameters used in the top-level cell, and can be added by the user for third-party purposes. Loss of cell properties will cause parameterized cells to lose the parameterization feature, but still behave as normal cells. Loss of the grid parameters may require the user to reset these manually. Files read with older versions will generate "`unsupported record type PLEX`" warnings in the log file if any of the new-style records are encountered.

B.2 The CIF File Format

The Cal (Tech) Intermediate Format (CIF) was developed at Cal Tech in the earliest days of design automation. The format, such as it is (there are many dialects), is public domain. Though possibly still used in educational and research environments, it is unusual in current commercial IC engineering.

The format used in native cell files and the device library file is an extension of the CIF file format. Through extension, this format is robust enough to meet the needs of *Xic* while retaining the syntactic simplicity of the original format. This section outlines the basic syntax of CIF, while the next section will provide details about the extensions used by *Xic*.

In CIF, "lines" are terminated with semicolons. The line feed and carriage return characters are taken as white space and ignored, and may not even be present, so the "lines" are actually logical only.

Comments in CIF are enclosed in parentheses. Comments are ignored in CIF, however *Xic* uses special comment lines for various purposes, as will be seen in the next section.

```
(This is an example comment);
```

Note that this (and all) CIF lines must be terminated with ';'.

The first one or two non-whitespace characters of a line (i.e., following ';') are used as a command key. In strict CIF, this key is a letter, though numbers have been adopted as widely-used extensions.

Historically, in CIF the word "symbol" has been used to refer to what in current terminology is referred to as a cell. When describing CIF, the terms "symbol" and "cell" are used as synonyms.

The `DS` (define symbol) directive begins a symbol (cell) definition.

`DS` *symnum A B*`;`

In strict CIF, symbols do not have names, but are referenced by symbol number. The assigned symbol number (an integer) follows `DS`. The remaining two parameters are for scaling. Each coordinate in the symbol is scaled by A/B. The use of two integers rather than a single floating point value was once considered a speed optimization. *Xic* never uses the *symnum* or scaling factors in native files:

```
DS 0 1 1;
```

The definition of a symbol is terminated with a `DF` line,

```
DF;
```

Within the symbol definition, there are layer directives, followed by geometry specifications, and subcell calls. A layer directive consists of a line with the form

`L` *layername*`;`

where *layername* is a name for a layer. In traditional CIF, the *layername* is an alphanumeric text token of four characters or fewer. All geometry which follows a layer declaration will be assigned to that layer, until the next layer declaration.

There are three types of CIF geometric objects used by *Xic*: boxes, polygons, and wires. Boxes have the form

`B` *width height x y* `[`*rx ry*`];`

where the first two parameters are the box width and height, and the second two parameters are the coordinates of the midpoint of the box. The last two parameters are optional, and indicate a rotation. The two numbers define a vector with respect to the origin, and the angle represents the angle by which the box should be rotated. *Xic* never uses the rotation parameters for boxes. Non-Manhattan rotated boxes are converted to polygons.

Ordinarily, boxes are rendered according to the attributes of the layer on which the box is defined. In *Xic* electrical mode, boxes on the SCED layer use that attribute, however boxes on other layers are rendered as a dotted outline with no fill. The SCED layer defaults to solid fill, other layers default to empty fill. All physical layers default to empty fill.

Polygons are specified with `P` followed by x-y coordinate pairs. The first and last coordinates must be the same, indicating closure of the polygon.

`P` *x0 y0 x1 y1 ... xN yN x0 y0*`;`

The polygon is rendered using the fill attributes of the layer on which the polygon is defined. There should be at least four pairs of coordinates defined for a polygon.

Wires are fixed-width paths. A wire is specified with `W` followed by the width, which is followed by x-y coordinate pairs representing the path.

W *width x0 y0 x1 y1 ... xN yN*;

In electrical mode, the basic line primitive is a zero width wire. In physical mode, wires are defined with finite width as a physical necessity. The coordinates will form the vertices of the path. A wire can technically consist of a single vertex, which will be rendered as a box with the width of the wire. This construct is disallowed in *Xic*, and should be avoided. Wires are rendered with the fill attribute of the layer on which the wire is defined.

A symbol call (subcell) is indicated with a `C`, followed by the symbol number, followed by a transformation specification. The transformation if made up of components representing translation, rotation, and reflection. Translation is indicated by `T` followed by the translation:

T *x y*

Rotation is specified by `R`, followed by two numbers which represent a vector with respect to the origin, the angle of which is the angle of rotation. Many parsers recognize only orthogonal rotations. *Xic* recognizes angles that are multiples of 45 degrees.

R *rx ry*

Mirroring about the y-axis is specified with

MX

and about the x-axis with

MY

The transformation specification is a concatenation of these directives, which are evaluated in sequence to obtain the coordinate mapping from the cell coordinates in the symbol being instantiated to the cell coordinates of the parent of the instance. The overall syntax is

`C` *symnum transform*`;`

where an example would be

```
C 0 R 1 0 T -1000 0;
```

The parsing is terminated with an end directive:

```
END
```

This line need not be terminated with a semicolon.

The base coordinate system specified for CIF uses 100 units per micron.

B.3 CIF Format Extensions

There have been numerous extensions to the CIF syntax used to enhance the capabilities of the original format. Some of these extensions have been accepted widely and have become essentially part of the standard. *Xic* uses these extensions, plus some further extensions, in native format files and in files converted to CIF without the **Strip For Export** button active. These extensions to the basic CIF syntax are enumerated below. Unless stated otherwise, the extension is applied identically in native cell files and CIF output.

When writing a cell hierarchy in CIF format, when the top-level cell is known, the writer will add a transform-less symbol call of the top-level symbol just before the final `End` line. Thus, the two final lines look like

`C` *top_cell_number* `;`
`End`

MOSIS specifically requires this. If the top level cell is unknown, which is true when translating directly from another format, this is skipped. *Xic* does not require or use this line.

1. Layer names can be arbitrarily long in CIF files generated or read by *Xic*, there is no four-character limit as in traditional CIF. In order to produce traditional CIF output, layer names should follow the traditional CIF limitation. Unlike some extensions, there is no provision for enforcing traditional CIF output, when layer names are arbitrary. Prior to release 3.3.0, *Xic* used CIF-style layer names.

2. If a semicolon is preceded by a backslash character, the reader will strip the backslash and propagate the semicolon as an ordinary character, and **not** as a record terminator. Thus, label and property strings may contain semicolons if they are “hidden” with a backslash.

3. Comment, label, and property strings can be arbitrarily long. Other interpreters, and older releases of *Xic*, may limit these lengths. Beware that the GDSII and CGX file formats have a 64KB record size limit and cannot accommodate strings that would overrun this limit (see the GdsTruncateLongStrings variable).

4. The `DS` (define symbol) line is always followed by a cell name extension line of the form
   `9` *symbol_name* `;`
   This extension is widely used, and is a standard means for including the symbol names within the CIF framework.

   In native cell files, however, the `DS` line is *preceded* by the symbol name line.

5. In *Xic* releases prior to 3.0.0, the symbol number part of an instance call was set to 0, i.e., the call sequence was always

   `C 0` ...`;`

   when cell name extensions were used. Since cell names were provided through the extensions, the cell numbering is unneeded. In current releases, the cell numbering is retained and will appear in the instance calls, in all CIF output.

6. In CIF, the name of the cell being instantiated may precede the “`C ...`” (symbol call) line, using the same format as associated with the `DS` line, i.e.

   `9` *master_name* `;`
   `C` N `...;`

   This is redundant in CIF, since the master name can be obtained from the symbol number. It is required in native cell files, as there is no symbol numbering.

   In native cell files only, instead of a cell name, the string can contain two words separated by white space. The first word is a path to an archive file (CIF, GDSII, etc.), library, or the name of an OpenAccess library. The second token would be the cell or library reference name. If the first word is a path to an archive file, the cell name is optional if the file contains only one top-level cell, this cell would be understood. When the cell is read into *Xic*, the master and its hierarchy will be read from the specified source.

   For example:

```
9 /path/to/directory/containing/myfile.gds mycell;
C 0 ...;
```

If a native cell with instance calls in this form is saved as a native cell, the instance calls will retain this special form.

Cell files that contain instance calls of this form can be produced in a couple of ways. One would first read the target archive cell into *Xic*, then create a new cell, and place one or more instances of the target cell. One can then write the new cell to disk as a native cell, and modify the instance calls with a text editor. Alternatively, one can use the rename function of the **Cells Listing** panel to change the name of the target cell to the two words as would appear in the instance call, i.e., for example the full path to the source file name followed by the cell name, separated by space. The rename will accept this form. Then write the current cell to a native cell file. There will be no need for text editing in this case. After doing this, however, you have a cell in memory with a bizarre name, best to clear the database or restart *Xic*.

7. Labels are specified with a unique syntax:
   `94 <<`*label string*`>>` *x y flags width height*`;`
   This is a further extension of a widely-used extension for labels, which does not have the *flags*, *width*, or *height* fields and the delimiters around the label. The original extension also required that the string contain no white space.

   The *width* and *height* are the dimensions of the untransformed bounding box of the label. The label will be stretched to fill this area. The label is surrounded by `<< >>`. The x and y are the reference coordinates, which by default is the lower left corner of the bounding box. The *flags* entry specifies transformations applied to the label at the reference point, and other rendering information. See C.2 for more information.

8. Cells and instances can be preceded by properties of the general form
   `5` *prop_num prop_string*`;`
   The property number *prop_num* is an arbitrary integer. The property string begins with the first non-space character following the integer, and ends with the semicolon (the semicolon is not included). The string can contain any alphanumeric, punctuation or white space but not ';' for obvious reasons. There are a number of properties used by *Xic*, particularly in electrical mode. This extension is widely used.

Xic writes the electrical information in a second symbol definition which immediately follows the physical cell definition in native files, but after the terminating token of the physical cell. Similarly, when *Xic* writes a CIF file without the **Strip For Export** function active, the electrical CIF representation immediately follows the physical CIF data, after the termination token.

In *Xic* releases prior to 3.0.0, the cell terminator was the single character `E`. This was used in both native cell files and unstripped CIF. In the present release, the cell terminator is always "`End`" in CIF, "`E`" in native cell files..

Whether or not these extensions are used when writing CIF output is controlled by a set of flags, which can be individually set from the **CIF** page of the **Export Control** panel. Actually, there are two banks of these flags, one bank is used when **Strip For Export** is set, the other bank is used when **Strip for Export** is unset. In the case of **Strip For Export** set, the flags all default to 0, so no extensions are used. In the case of **Strip For Export** unset, the flags all default to 1, so all extensions are used.

The user can set these flags individually through the **Extension Flags** menu in the **CIF** page of the **Export Control** panel. The bank of flags being set is determined by the state of the **Strip For Export** button and variable.

The flags in the menu have the following effects.

scale extension

Traditional CIF has a fixed resolution of 100 units per micron. This extension will add a comment of the form

`(RESOLUTION` *nnn*`);`

near the top of the file, and use *nnn* as the file resolution. The CIF reader must check for this comment and scale numerical values accordingly.

Xic normally uses internal units in unstripped CIF and native files, signaled with the addition of a comment line ahead of the first symbol definition something like:

`(RESOLUTION 1000);`

Xic will look for this comment, and interpret the coordinates accordingly. If no comment is found, the CIF default of 100 units per micron is assumed. *Xic* will always use internal units when writing a CIF file when this extension is enabled, and 100 units otherwise.

cell properties

Properties may be applied to cell definitions, ahead of the `DS`.

inst name comment

Comments of form

`(SymbolCall` *cellname*`);`

are added ahead of instance '`C`' calls.

inst name extension

Text in the form

`9` *cellname*;

is added ahead of instance '`C`' calls.

inst magn extension

Cell instance '`C`' calls can be preceded by a magnification extension of the form

`1 Magnify` *magn*;

where *magn* is a magnification factor. All internal structure of the cell will be scaled by the given factor, which is a floating point number greater than zero. This extension will appear in physical cell descriptions only. It is unique to *Xic*.

inst array extension

Cell instance '`C`' calls can be preceded by an array extension of the form

`1 Array` *x dx ny dy*;

where *nx* and *ny* are the number of cells to array in the x and y directions, and *dx* and *dy* are the spacing between cells. This extension was used in earlier CAD programs.

inst bound extension

Cell instance '`C`' calls can be preceded by a bounding box extension of the form

`1 Bound` *left bottom right top*;

The *left*, *bottom*, *right*, *top* are the coordinates of the parent cell defining the bounding box of the subcell. This extension is not currently used, though it is written into the files. It is unique to *Xic*.

inst properties
Properties may be added ahead ahead of instance 'C' calls.

obj properties
Properties may be added ahead of B (boxes), P (polygons), and W (wires).

wire extension
The end style of wires is not part of traditional CIF. In this extension, text of the form

`1 7033 PATHTYPE` *n*;

may be added ahead of wires to specify an end style. The values of *n* are 0 (flush ends), 1 (rounded ends), or 2 (extended ends, the default).

This extension was used in *Xic* prior to 2.5.23. It has been superseded by **wire extension new**, which will have precedence if both extensions are enabled.

wire extension new
This overrides **wire extension**, wires include an end-style designation:

`W0 | W1 | W2` *width x-y data*;

The end style of wires is not part of traditional CIF. In this extension, the wire end style 0-2 immediately follows the 'W', with the rest of the line as in traditional CIF. The end style is the same as the GDSII path type: 0 for flush ends, 1 for rounded ends, and 2 for extended square ends.

This extension was introduced in release 2.5.23. Older releases of *Xic* are not compatible with this extension.

text extension
Label string text is enclosed in << >>, and may include white space. Without this extension, white space characters in the label text will be replaced with underscores. In both cases, semicolons are replaced with underscores. This extension applies with any of the label format choices.

B.4 Native Cell File Format

Native cell files use a modified CIF format. Each file can contain two cell definitions: one for physical geometry, and the second for electrical (schematic) data. The physical information is found first. The electrical cell definition is optional, it may be absent or empty. An empty physical cell definition is created if there is no physical information, thus all native cell files produced by *Xic* will contain a physical cell definition. The parser will also recognize schematic files from the Jspice3 program, which have a similar format, but no physical cell definition.

The basic file layout is shown below.

```
(Symbol symbol_name);
(RCS_ID);
(program_version date);
(PHYSICAL);
(RESOLUTION 1000);
9 symbol_name;
```

```
DS 0 1 1;
physical data ...
DF;
E
(ELECTRICAL);
(RESOLUTION 1000);
9 symbol_name;
DS 0 1 1;
electrical data ...
DF;
E
SPICE listing
```

The first line is a mandatory CIF comment giving the symbol (cell) name. The second line is an optional comment providing an ID field to be used with the RCS/CVS code control programs. These programs are used to manage large projects with multiple designers. Other comment lines may follow, in particular a comment line containing the creating program version and creation date is added by *Xic*. The next line is a CIF comment containing the word "PHYSICAL". This indicates that the following cell definition contains physical data. If this line is not found, some time consuming tests are performed to figure out what exactly is in the file.

An optional "`(RESOLUTION 1000);`" comment line follows. This indicates that coordinates in the physical part of the file use 1000 units per micron. If the line is not present, 100 units per micron is assumed. This was the default for early versions of *Xic*, and follows from the implicit CIF assumption. The use of resolutions other than 100 represents an extension of the CIF syntax.

The integer following "`RESOLUTION`" can be 100 or any of the values supported for the DatabaseResolution variable, for the physical cell. For the electrical cell, only the values 100 and 1000 are allowed.

The electrical part of the file is optional, and starts with a CIF comment containing the word "ELECTRICAL", followed by the resolution comment and the electrical cell description. Either cell description can be empty, i.e., a `DS 0 1 1;` line followed by `DF;` and `E`. Finally, if the cell was written in schematic mode and is a top-level cell (containing no terminal nodes) a SPICE listing of the circuit is added to the bottom of the cell file. Such files can be read directly into the *WRspice* program for simulation. The SPICE listing has no relevance to *Xic*.

In release 3.1.5 and later, the terminating line of a native cell file can have "`n`" or "`nd`" (case insensitive) following the "`E`", as in normal CIF. In earlier releases, anything after "`E`" would cause a syntax error.

The format of the Physical cell data adheres to the extended CIF described in the preceding sections. Electrical cell descriptions use the same extensions, however the array extension never appears, as arrays are not available in electrical mode. The major difference in the files is the large number of properties assigned in electrical mode.

B.5 Computer Graphics Exchange (CGX) Format

The Computer Graphics eXchange (CGX) format is a simple binary data format somewhat similar to GDSII, but designed to be more compact. Like GDSII, files consist of a sequential list of variable-length records. It has simplified record structure, but extensions in data flexibility. If is more compact than GDSII and is more efficient to read and write.

The advantages of CGX are smaller files and faster read/write than GDSII. This format was developed by Whiteley Research Inc., but is hereby placed in the public domain without restriction.

The file extension is "`.cgx`". Gzipped files ("`cgx.gz`") are supported. *Xic* will automatically identify this file type, and can read, write, and convert to *Xic* files just as GDSII.

B.5.1 CGX Format Identifier

The first three bytes of a CGX file are 'c', 'g', and 'x'. The fourth byte is an integer format level. A parser designed to handle a certain level will accept that level and any value lower. Presently, the only existing level is 0, thus this byte should be set to 0.

B.5.2 CGX Data Types

CGX uses the same long (4–byte) and short (2–byte) integer formats as GDSII, and the same 8–byte floating point format. These are the only numerical data types defined.

A date is stored as 8 bytes, as shown in the following table. These are the same numerical fields as used in GDSII, though the format is different (bytes are used where possible, rather than shorts). The third column gives the value in terms of the members of the `tm` structure from the C library.

short	year	`tm_year + 1900`
byte	month	`tm_mon + 1`
byte	day	`tm_mday`
byte	hour	`tm_hour`
byte	minute	`tm_min`
byte	second	`tm_sec`
byte	0	

Strings are stored in the same manner as in GDSII. The null terminator in not written, however a null byte will be added to strings of odd length, so that record sizes are always even.

B.5.3 CGX Data Records

The four–byte file header is followed by any number of data records, the last of which signals the end of data. There are 11 defined record types. Each record begins with a 4-byte header:

short	`recsize`
byte	`rectype`
byte	`flags`

The `recsize` field is a short unsigned integer giving the total record size, including the header. Thus, as in GDSII, records are limited to 64K bytes in length. The record size will always be an even number. The `rectype` byte is set to a small integer to define the type of record. The `flags` byte is used in some of the record types, otherwise it is ignored.

The defined record types are given in the table below.

rectype	**name**
0	LIBRARY
1	STRUCT
2	CPRPTY
3	PROPERTY
4	LAYER
5	BOX
6	POLY
7	WIRE
8	TEXT
9	SREF
10	ENDLIB

It is allowable to define additional record types for local or proprietary purposes. If a parser encounters an unknown record type, it may skip over the record, ignoring it.

`LIBRARY` record The `LIBRARY` record should be the first data record in the file, and can appear once only.

The `flags` byte of the record header can be used for a version number, which identifies in some way the remaining data in the file.

The `LIBRARY` record contains the following fields:

bytes	**field name**	**purpose**
8	`munit`	machine units
8	`uunit`	user units
8	`cdate`	library creation date
8	`mdate`	library modification date
?	`libname`	library name string

The first two fields are double-precision numbers that define the scale factors for the data in the file. These are interpreted in the same way as the similar fields in the header of a GDSII file.

The second two fields represent creation and modification dates for the file content.

A name string for the library follows. Strings are null-byte terminated, and an additional null byte is added if necessary so that the total length is even.

`STRUCT` record The `STRUCT` record opens a cell structure. Records that follow will be assigned to that cell, until another `STRUCT` record is seen.

The header `flags` byte is not used.

The `STRUCT` record contains the following fields:

bytes	**field name**	**purpose**
8	`cdate`	creation date
8	`mdate`	modification date
?	`strname`	structure name string

The first two fields provide creation and modification dates for the structure. These are followed by a string giving a name for the structure. This name should be unique in the file.

`CPRPTY` **record** Zero or more `CPRPTY` records can appear following a `STRUCT` record. These are properties that are applied to the cell.

The header `flags` byte is not used.

The `CPRPTY` record contains the following fields:

bytes	**field name**	**purpose**
4	`number`	property number
?	`string`	property string

Any number or string is allowed.

`PROPERTY` **record** Zero or more `PROPERTY` records can appear ahead of `BOX`, `POLY`, `WIRE`, `TEXT`, and `SREF` records. It assigns a property to the object that follows.

The header `flags` byte is not used.

The `PROPERTY` record contains the following fields:

bytes	**field name**	**purpose**
4	`number`	property number
?	`string`	property string

Any number or string is allowed.

`LAYER` **record** A `LAYER` record can appear after a `STRUCT`, and must appear before any of `BOX`, `POLY`, `WIRE`, `TEXT` in the `STRUCT`. The layer context will persist until the next `LAYER` or `STRUCT` record.

The header `flags` byte is not used.

The `LAYER` record contains the following fields:

bytes	**field name**	**purpose**
2	`number`	layer number
2	`datatype`	data type
?	[`lname`]	optional layer name

The layer number and data type are sufficient, and have the same interpretation as in GDSII. Alternatively or in addition, a string giving a layer name can be supplied.

`BOX` **record** A `BOX` record can appear after a `LAYER` record has been issued. The `BOX` record defines one or more rectangular data objects.

The header `flags` byte is not used.

The `BOX` record contains the following fields:

bytes	**field name**	**purpose**
4	`left`	left value
4	`bottom`	bottom value
4	`right`	right value
4	`top`	top value
?	[`repeat`]	repeat for multiple boxes

The first four integers define a box, and a record can contain multiple box definitions (four integers per box). Each box is given the properties currently in effect, and is assigned to the layer currently in effect.

POLY record A POLY record can appear after a LAYER record has been issued. The POLY record defines a polygon object.

The header `flags` byte is not used.

The POLY record contains the following fields:

bytes	field name	purpose
?	`xy`	coordinate pairs, path must be closed

Coordinates use four-byte integers. The first and last coordinate pair (x–y values) must be the same. There must be at least four coordinate pairs.

WIRE record A WIRE record can appear after a LAYER record has been issued. A WIRE record specifies a single wire (path) data object.

The header `flags` field contains a value in the range 0–2 which sets the end style of the wire:

0	flush ends
1	rounded ends
2	extended square ends

This is the same as the pathtype in GDSII.

The WIRE record contains the following fields:

bytes	field name	purpose
4	`width`	path width
?	`xy`	coordinate pairs (1 pair or more)

TEXT record A TEXT record can appear after a LAYER record has been issued. A TEXT record specifies a non-physical text object.

The header `flags` byte is an orientation code:

bits 0-1	rotate the text about the anchor
	00 no rotation
	01 90 degrees
	10 180 degrees
	11 270 degrees
bit 2	mirror y after rotation
bit 3	shift rotations to 45, 135, 225, 315 degrees
bits 4-5	horizontal justification, 00 left, 01 center, 10,11 right
bits 6-7	vertical justification, 00 bottom, 01 center, 10,11 top

Note that this is identical to the lowest byte of the *Xic* label flags (see C.2).

The TEXT record contains the following fields:

bytes	field name	purpose
4	*x*	x position
4	*y*	y position
4	*width*	field width
?	*label*	label text
1	*flags*	additional flags

The *width* gives the physical equivalent width of the text. The height is determined by the font used for rendering.

In order to accommodate additional flags, a flags byte is "hidden" behind the label text. The flags byte follows the null byte that terminates the text string. A null byte may be added following the flags byte to make the total byte count even. The flag bits are:

`SHOW`	0x1
`HIDE`	0x2
`TLEV`	0x4
`LIML`	0x8

These flags are described with the XprpXform pseudo-property in 10.1.2.

`SREF` record The `SREF` record describes an instance, or an array of instances.

The header `flags` byte can have any of the following bits set.

`ANGLE`	0x1
`MAGN`	0x2
`REFLECT`	0x4
`ARRAY`	0x8

The `SREF` record contains the following fields:

bytes	field name	purpose
4	`x`	x coordinate
4	`y`	y coordinate
8	`angle`	rotation angle, if `ANGLE` flag only
8	`magnif`	magnification, if `MAGN` flag only
4	`cols`	array columns, if `ARRAY` flag only
4	`rows`	array rows, if `ARRAY` flag only
16	`xy[4]`	aref points (like GDSII), if `ARRAY` flag only
?	`sname`	referenced structure name

If the `ANGLE` flag is set, the cell is to be rotated by an angle, in degrees, found in the record. If the `MAGN` bit is set, the cell is scaled by a value found in the record. If the `REFLECT` bit is set, the instance will be reflected about the x–axis, as in GDSII. If the `ARRAY` bit is set, the instance is arrayed, as in GDSII, where x, y, and xy give the three orientation points, as in a GDSII AREF record. Unless the corresponding bit is set, the corresponding data are not in the record.

`ENDLIB` record The `ENDLIB` record must be the last record of the file. It contains no data.

B.6 OASIS Format

As integrated circuit mask layouts inexorably increase in complexity, the fundamental limitations of the industry standard GDSII file format have become a bottleneck. A major weakness of the GDSII format is inefficient data representation, which leads to very large files. File sizes of tens of gigabytes are not uncommon, leading to difficulties in transmission, data integrity, and consumption of hardware resources.

The Open Artwork System Interchange Standard (OASIS) was designed by the SEMI consortium (`http://www.semi.org`) as a modern alternative to the GDSII standard. A draft specification (SEMI Document 3626 2003/04/23) of the OASIS format standard was circulated, and subsequently adopted with very minor changes (SEMI P39-1105). The final standard document is available from the SEMI organization.

The main objective of the OASIS standard is efficient representation of mask layout geometry, both in hierarchical and flat representations. The format makes use of a number of techniques to this end.

- A compact variable-size integer representation is used. Along with heavy use of offsets, one and two byte integers can be used extensively in place of the larger fixed-size integers used in other formats.
- Extensive use of modal variables greatly reduces repeated information.
- String and name referencing by number eliminates repetition of these data.
- A flags byte indicates the presence or absence of certain data fields in most records, so that unused or unset values do not need to be included in the stream.
- Special compact representations for trapezoids and other common features save space.
- An encoding mechanism for repetitions can be used to consolidate arrays of objects.
- A data compression mechanism is supported.

As a "typical" example, the sizes in the table below illustrate the space-saving capability of the OASIS format. This lists the size of a GDSII file, and the size of the resulting OASIS file as converted with *Xic* with the main available options.

File	**Size (bytes)**
GDSII file	7669760
OASIS, basic	1643804
plus repetitions	1153071
plus name tables	1067157
plus compression	816225

B.6.1 OASIS Support in *Xic*

Xic was the first (to our knowledge) commercial implementation of the OASIS format. Some limited tools have been made available from Mentor Graphics (GDSII/OASIS translator), and SoftJin (GDSII/OASIS translator and text mode converters). We recommend `anuvad` from SoftJin (`http://www.softjin.com/html/anuvad.htm`), which has been our "reference" in establishing portability.

This capability was designed from the draft SEMI-3626 document, but has incorporated changes from the final specification.

This section describes the OASIS capabilities in *Xic*. The present status of OASIS support in *Xic* is complete, the bottom line being

1. *Xic* can read any spec-conforming OASIS file.
2. OASIS output from *Xic* is readable by any other spec-conforming tool.
3. Exceptions to the above are **bugs**, please report!

OASIS is one of the supported archive formats, along with GDSII, CIF, and CGX. CGX (Computer Graphics eXchange) format is another "improved" GDSII developed and placed in the public domain by Whiteley Research. The archive formats have the following capabilities in *Xic*:

- Files can be read directly into *Xic*, either using the **Open** command, or with similar buttons and functions in *Xic*.
- Files can be converted directly to another (or the same) archive format, or to *Xic* native cell files, from the **Format Conversion** function in the **Convert Menu**. While converting, scaling, windowing (clipped or not) and flattening can be employed. There is also provision for selecting the layers to convert.
- *Xic* can output a hierarchy in memory to any of the archive formats. The default format is the format of origin, if any.
- The random access of cells from the file, such as with the **Contents** function of the **Files Listing** or the library access mechanism applies to all archive formats.
- The Cell Hierarchy Digest (CHD) and Cell Geometry Digest (CGD) features, which facilitate working with very large files (too large to fit into the main memory database) apply to all archive formats.
- The script function that splits a file spatially into pieces, `ChdWriteSplit` applies to all archive formats for both input and output.

B.6.2 Characteristics of OASIS Output From *Xic*

The basic OASIS file generated by *Xic* has the characteristics listed below.

- By default, all strings are saved locally as strings, i.e., no indirection is used, so there are no `<name>` records. This can be changed with the OasWriteNameTab variable which is connected to **Use string tables** check boxes in the **Format Conversion** and **Export Control** panels.
- By default, no REPETITION records are generated for `<geometry>` records. If the OasWriteRep variable or the corresponding check box is set, REPETITION record types may be generated. This option attempts to recognize arrays of identical objects when writing OASIS files.
- By default, appropriate three and four-sided polygons will be written as TRAPEZOID or CTRAPEZOID records, however this can be disabled with the OasWriteNoTrapezoids variable.

- By default, wires (paths) will retain that data type. However, rectangular two-vertex paths will be converted to a more compact rectangle representation if OasWriteWireToBox is set.

- The following record types are not generated by *Xic*: CIRCLE, XNAME, XELEMENT, XGEOMETRY.

- When writing OASIS files with StripForExport set, i.e., writing physical data only, and when using string tables, the offset table is placed in the END record. With StripForExport not set, in general we write two sequential OASIS databases into the file, the first for physical data, the second for electrical. This is a *Xic*-specific extension. In this case, string tables are used in the physical part only, and the offset table is placed in the START record. PAD records are added to avoid overwriting data since this is a non-sequential operation. In all cases, strict-mode tables are used.

  Note: If a design contains physical data only, the electrical records are absent, so that the file becomes conventional. Even if electrical records are present, the reader will probably ignore them (as does `anuvad-0.2`). However, when exporting physical data, for portability StripForExport should always be set.

  The string tables themselves are written just ahead of the END record in all cases (when tables are used).

- OASIS files generated by *Xic* release 3.2.2 and later have a file property named "XIC_SOURCE", with no content. This identifies the file as originating from *Xic* or a derivative.

- All integer values are 32-bit limited, except for values that represent offsets into the file, which may be 64-bit.

- The OASIS format does not provide a native code to indicate a rounded-end wire. For wires that have rounded ends, i.e., that originated as GDSII PATHTYPE=1, the half-width extension is specified, and the PATH record is given an empty (info byte = 0x4) property named "XIC_ROUNDED_END".

- The OASIS format does not provide codes for TEXT element presentation. In *Xic*, these are used for on-screen labels, and are treated by *Xic* as any other database object, but they will not appear on the mask layout. Thus, at least for *Xic* internal use, TEXT presentation attributes are important. They are stored in a property applied to TEXT records named "XIC_LABEL". The XIC_LABEL property contains two unsigned integers. The first is the width of the label, in database units. The second is the label flags word used by *Xic* which specifies many presentation attributes. See C.2 for more information.

- OASIS text labels can contain only printable ASCII characters and the space character, thus some trickery is used to support multi-line labels. In OASIS files generated from *Xic*, the following non-printing characters are replaced with the indicated character sequence when encountered in label text:

`0xa` (line feed)	"`\n`"
`0xd` (carriage return)	"`\r`"
`0x9` (tab)	"`\t`"

  The OASIS reader will perform the reverse conversion, if the XIC_SOURCE property is found in the file, meaning that it was written by *Xic*.

- All other properties, which might be given to CELL or `<geometry>` records, are named "XIC_PROPERTY" and consist of concatenated number/string pairs. *Xic* uses properties indexed by a number, with string-type data, so that the XIC_PROPERTY consists of the list of properties as known to *Xic* for that cell or object.

There are several options in *Xic* that modify OASIS input/output. Many of these can be controlled by check boxes in the **OASIS** page in the **Export Control** and **Format Conversion** panels from the **Convert Menu**, which reflect the status of the variables (which can also be set with the **!set** command or equivalent).

Convert Menu - Input and ASCII Output

OasReadNoChecksum	Ignore checksum in OASIS input file
OasPrintNoWrap	Use one line per record in OASIS ASCII output
OasPrintOffset	Add file offsets to OASIS ASCII output

Convert Menu - Output

OasWriteCompressed	Compress records in OASIS output
OasWriteNameTab	Use string table referencing in OASIS output
OasWriteRep	Try to combine similar objects in OASIS output
OasWriteChecksum	Compute and add checksum to OASIS output
OasWriteNoTrapezoids	Don't convert polys to trapezoids
OasWriteWireToBox	Convert wires to boxes when possible
OasWriteNoGCDcheck	Don't look for common divisors in repetitions
OasWriteUseFastSort	Use faster but less effective sorting
OasWriteNoXicTextPrps	Don't write certain text properties

B.6.3 Requirements And Limitations for Reading OASIS

Xic can very likely read any OASIS file that meets the published specification. Exceptions should be reported as bugs!

- Properties are ignored, unless the name matches one of those understood by *Xic* (see above). The file properties set an internal variable but otherwise do nothing.
- The XNAME, XELEMENT, and XGEOMETRY records are ignored.
- The CIRCLE record will create a polygon object approximating a circle, with the number of sides using the internal variable in *Xic*.
- The TRAPEZOID and CTRAPEZOID records will create a polygon object.
- The REPETITION records found in PLACEMENT records will define a cell array if possible (i.e., represents a periodic Manhattan configuration), otherwise individual cell instances will be created and replicated. In `<geometry>` records, any REPETITION is accepted, but the repetition is decomposed and separate objects are created in memory.

B.7 Library Files

Library files are *Xic* input files which contain references to cells, other libraries, or cell definitions. The format of a library file is as follows:

`(Library` *libname*`);`
`#` *any comments*

`#` *optional keywords to implement conditional flow*

```
Define [eval] name [value]
If expression
IfDef name
IfnDef name
Else
Endif

Property number string
...
Alias alias refname
...
Reference name path [cellname]
...
Directory path
...
(Symbol symname);
symbol definition
E
...
```

The first line must begin with "`(Library `", which designates a library file to *Xic*. The *libname* on this line following `Library` is ignored, but by convention is the library file name. Within the file are three kinds of data fields: properties, references, and cells. Any line starting with a pound sign ('#') is taken as a comment and ignored. Blank lines are ignored.

It is recommended that library files be given a "`.lib`" extension. This is not a strict requirement, except that the listing of libraries from the search path provided in the **Libraries List** button in the **File Menu** will contain only files with this extension.

All library files (including the device library) support a limited macro capability. The macro capability makes use of the generic macro preprocessor provided in *Xic*, which is described in 18.1. The reader should refer to this section for a full description of the preprocessor capabilities. The preprocessor provides a few predefined macros used for testing (and customizing for) release number, operating system, etc. The keyword names, which correspond to the generic names as described for the macro preprocessor, are case-insensitive and listed in the following table.

Keyword	**Function**
`Define`	Define a macro.
`If`	Conditional evaluated test.
`IfDef`	Conditional definition test.
`IfnDef`	Conditional non-definition test.
`Else`	Conditional else clause.
`Endif`	Conditional end clause.

These can be used to conditionally determine which parts of the file are actually loaded when the library is read. Presently, there is no macro expansion or text substitution in lines of text in the library, the macros simply implement flow control. Otherwise, they work the same as similar keywords in the technology file (see A.2) and in scripts (see 18.8), and are reminiscent of the preprocessor directives in the C/C++ programming language.

Properties are used in the device library file (which is a special library file which must exist in order to use electrical mode), and are described in the description of the device library file format.

Aliases provide alternative names by which data records can be obtained from the library. In particular, for the device library, this facilitates accessing library devices under alternative names. For example, in older device libraries, the terminal device was named "`vcc`", while the present name of a similar terminal is "`tbar`". The addition of

```
Alias vcc tbar
```

will satisfy references to the `vcc` terminal device in older designs.

References associate a name with a cell, or another library. For a cell, *name* (above) is the name by which the cell will be added to the database when opened, and the name that will appear in selection listings. The *path* is a path to the file containing the cell, which can be native (*Xic*), or a path to an archive file containing the cell. If the path contains white space characters, it should be single or double quoted.

Aliases may be used to provide alternative names.

If the path points to an archive file, the *cellname* argument can be set to the name of the cell in the file. Note that this does not have to be the same as *name*. Opening *name* will open the cell referenced and add it to the database as *name*. Any subcells that have references in the same library file will be opened under the library reference name. All other cell name aliasing is suppressed, except for AutoRename.

If *cellname* is not given, opening a reference to an archive file with multiple cells will cause a pop-up to appear, allowing the user to choose which cell to open. In this case, the cell will be opened under its own name.

If *path* points to another library file, then *cellname*, if given, indicates which reference in the library to open, i.e., it should be one of the *name*s in the referenced library. In this case, the cell will be opened as *name* in the original library. If *cellname* is not given, a pop-up will appear allowing the user to choose which library element in the referenced library to open. A cell selected in this way will be opened as *name* in the referenced library. Thus the `Reference` keyword provides a means for multiple-level indirection through the library files.

The `Directory` keyword is followed by a full path to a directory. Every layout or library file found in the directory is logically added as a `Reference`, but with no *cellname* given. This keyword provides an easy way to reference a collection of cell files, for example.

Cells can be defined within libraries by including the native-format body in the library file. The first line of the cell must start with "`(Symbol `". The symbol text should contain both the electrical and physical blocks. The cells in the device library file are special in that they contain only an electrical block, so are not representative. Cells can be added to a library with a text editor, by copying from native cell files. The name of the cell is actually given by the lines with format like "`9` *symbolname*;" and the *symbolname* in the "`(Symbol` *symbolname*`);`" is actually ignored. The user need not concern themselves with details of the format, it is sufficient to simply copy the entire *Xic* cell file into the library, however any trailing SPICE listing should be excluded, including the "`*Generated by Xic`..." line.

B.7.1 Example Library File

The example below uses the `Reference` directive only, which is common. It illustrates some of the types of references that are possible.

```
(Library demo.lib);
```

```
# simple reference to native cell
Reference acell           /usr/.../xic_cells/acell

# simple reference to native cell, with name change
Reference buffer          /usr/.../xic_cells/cell32

# browsable reference to a GDSII file
Reference gdsfile.gds     /usr/.../gdsfile.gds

# reference to cell in GDSII file, with name change
Reference mux8_1          /usr/.../gdsfile.gds      MUX

# reference to cell in CIF file
Reference and5            /usr/.../ciffile.cif      and5

# browsable reference to another library
Reference stdcells_25.lib /usr/.../stdcells_25.lib

# indirect references to cells in another library
Reference orgate_25       /usr/.../stdcells_25.lib  orgate
Reference andgate_25      /usr/.../stdcells_25.lib  andgate

# Reference to all layout files and libraries found in a directory
Directory                 /home/joe/devices
```

B.8 Device Library File

The device library file is a special library file which contains all of the information required to render and otherwise support the devices available in the electrical mode of *Xic*. It is expected to be found along the library search path. The search is always performed in the current directory first, whether or not this is indicated by the search path. The default name for this file is "`device.lib`", however this name can be changed with the `DeviceLibrary` keyword in the technology file. Only one device library is used, and the first file found in the search path with a matching name is read.

In present *Xic* releases, devices are not required to be supplied in the device library, however only devices referenced in the library will appear in the device selection menus. When loading a design produced in Virtuoso, for example, the schematic symbols are imported as cells which function as devices, but are not automatically included in any device library. As long as *Xic* can find these device cells when the layout file is loaded, all is well. This is automatic if the device cells are kept in the layout files, which is true by default. However, the device cells can be added to the device library in various ways, as described below. This has the following advantages:

1. The devices are available in the device menus available from the side menu in electrical mode. This facilitates use of the device in other designs.
2. The devices are no longer included in the layout files, reducing their size. However, the library must be provided along with the files when exporting the design.

Devices can be either primitive devices as used by SPICE, or subcircuit macros. If the device represents a subcircuit macro, the name of the subcircuit is given as a model property, and that subcircuit must exist in one of the model library files. For example, a device named "opamp" could be added to the device library file. Then the user would set the model property to something like "ua741" which would have a subcircuit definition somewhere in the model library files (perhaps in a directory containing SPICE models obtained from a semiconductor manufacturer).

There are four classes of device that may appear in the device library file. The first class consists of basic elements such as resistors, capacitors, and semiconductor devices which have physical implementations in a layout and are known elements in SPICE. The second class consists of voltage and current sources, which are known elements in SPICE but do not have physical equivalents in a layout. The third class applies to macros, which expand to a subcircuit in SPICE. These may or may not have an actual physical embodiment. The fourth class are terminals, which are used in the electrical schematic to provide connections. These are not used in SPICE, but are used to establish connectivity when producing SPICE input. They have no direct physical implementation, but imply physical connections.

The first line of the file must be in the form

`(Library` *filename*`);`

This is the signature used in all library files.

Comment lines, which are ignored when the file is parsed, begin with the '`#`' character, and can appear anywhere outside of the device definitions except on the first line. Lines containing only white space are ignored.

B.8.1 Device Library Global Properties

The device library file handles "global properties". These properties appear at the beginning of the file, after the initial line but before the definitions. The syntax is

`Property` *identifier string*

where `Property` appears literally, *identifier* is a keyword or equivalent integer as described below, and the rest of the line constitutes the *string*. There can be any number of these lines.

The following properties are recognized:

`SpiceDotSave` This property is identified by the keyword "`SpiceDotSave`" or by the integer 20.

The *string* consists of a SPICE key letter for a device (such as '`R`' for a resistor), followed by the name of a parameter known to SPICE for that device. While a SPICE deck is being created, and if this property was given, each device in the circuit that is keyed by that letter will trigger the addition of a line in the SPICE file in the form

`.save @`*name*[*param*]

The *name* is the name of the device, and the *param* is the parameter name given in the property. This construct forms a vector name which the directive ensures will be saved during simulation, and thus be available for output. This is the means by which device parameter data are made available *by default* in SPICE runs initiated from SPICE output generated by *Xic*. *WRspice* and other SPICE3-derivative simulators will recognize this form, however only *WRspice* will actually save the vector in interactive mode. SPICE3 ignores `.save` lines, except in batch mode.

This property is used in the supplied `device.lib` file, for current sources and the "`c`" (current) parameter. The **branch** property for current sources references "`@`*name*`[c]`", so that it is important to ensure that this vector is saved. Thus, the appropriate global property is

`Property SpiceDotSave I c`
or equivalently
`Property 20 I c`

This will produce lines in the SPICE output like

```
.save @isource[c]
```

for a current source named `isource`.

`DefaultNode`

This property is identified by the keyword "`DefaultNode`" or by the integer 21.

This property is used for providing a default node name for the last node listed in a SPICE output device line. This allows the use of a three-node MOS device, with the substrate node connected automatically. The feature is enabled by adding the following property line at the top of the device library file:

`Property DefaultNode` *device_name num_nodes node_name*
or equivalently
`Property 21` *device_name num_nodes node_name*

The parameters are:

device_name:	name of device (e.g., `nmos`)
num_nodes:	number of nodes expected by SPICE
node_name:	name of node to be added

For example,

```
Property 21 nmos 4 NSUB
```

A `Property` line should be added for each device which has a default node. The respective device descriptions in the device library file should also be modified to remove the substrate mode. The supplied `device.lib` file contains MOS models with this feature included, and also standard models.

Using the example above, a SPICE output deck will contain lines like

```
M1 1 2 3 NSUB ...
```

Also, there will be a line added at the top of the deck:

```
.global NSUB
```

This line tells *WRspice* to not modify this node name during subcircuit expansion. The user must explicitly add a connection to the global node, usually to a voltage source. This can be accomplished in *Xic* by placing a terminal device, and modifying the terminal name to the node name (NSUB).

`DeviceKey`

This property is identified by the keyword "`DeviceKey`" or by the integer 22.

Although still recognized, this property is obsolete and should not be used. The `DeviceKeyV2` property syould be used instead.

There is an internal table of mappings from letters to devices, in accordance with the definitions and traditions of SPICE. For example, '`r`' (case insensitive) maps to a resistor device. It is possible to define new device keys, overriding the defaults. It is also possible to define multi-letter keys.

These keys apply when *Xic* reads a SPICE file and maps devices to those found in the `device.lib` file.

The format for the property specification is

> `Property DeviceKey` *prefix opt val nnodes nname pname*
> or equivalently
> `Property 22` *prefix opt val nnodes nname pname*

prefix
: This is a short (usually single-character) device identification prefix, the first character if which must be a letter.

opt
: This is a binary value, the token can be `0`, `no`, or `off` if unset, or `1`, `yes`, or `on` if set. If set, then the presence of the last connection node of the device is optional (such as for a BJT, which has an optional substrate node).

val
: This is a binary value as above. If set, text following the nodes is saved in a **value** property and the **model** property is unset, as for voltage/current source devices.

nnodes
: An integer giving the number of device nodes, including the optional node if any.

nname
: The device name, or the n-type device, in the library.

pname
: If this is not 0 or missing, it is the name of the p-type library device.

`DeviceKeyV2`

This property is identified by the keyword “`DeviceKeyV2`” or by the integer 23.

This is an extended version of the now-obsolete `DeviceKey` property that supports current-controlled sources and switch, and will allow any number of optional nodes. The older format is still recognized, but can not be used to create standard keys for these devices.

The format for the property specification is

> `Property DeviceKeyV2` *prefix min max devs val nname pname*
> or equivalently
> `Property 23` *prefix min max devs val nname pname*

prefix
: This is a short (usually single-character) device identification prefix, the first character if which must be a letter.

min
: This is the minimum number of nodes used by the device.

max
: This is the maximum number of nodes used by the device. The difference from the *min* is the number of optional nodes.

devs
: This is the number of device reference names, which is one for current-controlled sources and switch.

val

This is a binary value, the token can be `0`, `no`, or `off` if unset, or `1`, `yes`, or `on` if set. If set, text following the nodes is saved in a **value** property and the **model** property is unset, as for voltage/current source devices.

nname

The device name, or the n-type device, in the library.

pname

If this is not 0 or missing, it is the name of the p-type library device.

The internal table provides the following defaults.

prefix	*min*	]it max	*devs*	*val*	*nname*	*pname*
`a`	2	2	0	true	`vsrc`	0
`b`	2	3	0	false	`jj`	0
`c`	2	2	0	false	`cap`	0
`d`	2	2	0	false	`dio`	0
`e`	2	4	4	true	`vcvs`	0
`f`	2	2	1	true	`cccs`	0
`g`	4	4	0	true	`vccs`	0
`h`	2	2	1	true	`ccvs`	0
`i`	2	2	0	true	`isrc`	0
`j`	3	3	0	false	`njf`	`pjf`
`k`	0	0	2	false	0	0
`l`	2	2	0	false	`ind`	0
`m`	4	4	0	false	`nmos`	`pmos`
`n`	0	0	0	false	0	0
`o`	4	4	0	false	`ltra`	0
`p`	0	0	0	false	0	0
`q`	3	4	0	false	`npn`	`pnp`
`r`	2	2	0	false	`res`	0
`s`	4	4	0	false	`sw`	0
`t`	4	4	0	false	`tra`	0
`u`	4	4	0	false	`urc`	0
`v`	2	2	0	true	`vsrc`	0
`w`	2	2	1	false	`csw`	0
`x`	0	0	0	false	0	0
`y`	0	0	0	false	0	0
`z`	3	3	0	false	`nmes`	`pmes`

The user working with MOS technology may need to understand and set this property for "`m`" (MOS) devices. For LVS, is is required that the electrical and physical MOS devices assume the same number of nodes. The device library provides a choice of three-terminal (**nmos**, **pmos**) and four-terminal (**nmos1**, **pmos1**) devices. Although either type of device can be placed in a schematic that is used for simulation, for comparison to the physical layout consistency is required with the MOS device extraction templates defined in the technology file `Device` blocks (see 16.8.1).

For consistency, there are two choices:

1. The technology defines a three-terminal "nmos" device, and the schematics exclusively use the **nmos** schematic symbol (similar for pmos). In this case, substrate/well connectivity is simply ignored in comparisons.

2. The technology file defines a four-terminal “nmos1” device, and the schematics use the **nmos1** schematic symbol (similar for pmos). In this case, the substrate/well connection at each transistor is included in the connectivity comparison.

Three and four terminal devices of the same sex can not be mixed in physical extraction, however they can be different for p and n devices. For example, in a process where only the pmos devices reside in a defined “tub”, it might be convenient to use three-terminal nmos devices, and four terminal pmos devices. In this case, the technology file should define extraction devices for a three-terminal “nmos”, and four terminal “pmos1”. The standard `device.lib` file should include the line

```
Property DeviceKeyV2 m 4 4 0 false nmos pmos1
```

and the user should remember to use the three-terminal **nmos** and four-terminal **pmos1** library devices exclusively in schematics that will be used with physical data.

The **!devkeys** command dumps the current keys to the console window, which can be useful for debugging this capability.

B.8.2 Device Library Aliases

The device library may use the `Alias` keyword

`Alias` *alias libcellname*

to define alternate names for devices contained in the library. The alternate names can be used equivalently when referencing devices from the library. Aliases, however, will **not** appear in the device menu displayed from the electrical side menu in *Xic*

The `device.lib` file distributed with *Xic* provides aliases to terminal devices whose names have been changed from those used in earlier *Xic* releases, thus providing backward compatibility. The device names were changed in release 3.2.22.

old name	**current name**
`vcc`	`tbar`
`vbus`	`tbus`

B.8.3 Device Library Devices

A device is simply an electrical cell definition. It is distinguished as a device by the presence and values of certain properties. Devices have no subcells or sub-devices, they must contain only geometry. They can not contain physical data.

The supplied `device.lib` file contains a collection or rather plain looking generic device models that correspond to the devices supported by SPICE. Additional devices are often created when a layout is imported from Cadence Virtuoso. These correspond to the schematic symbols of devices used in Virtuoso schematics. Devices can be created by the user through use of the **Save As Device** button in the **File Menu**. These can be used to supplement or replace devices provided in the default library.

Device definitions as native cell definitions, whether inline in the `device.lib` file, or in separate files, may or may not have an empty physical part. This is in contrast with normal cells, where a physical part is required. In releases prior to 4.1.12, inline cell definitions could not have an empty physical block, but in a cell file the empty physical block was required.

Devices can be referenced in three ways:

1. By using the `Reference` keyword to reference a cell containing the device definition.
2. By using the `Directory` keyword to reference a directory that contains device definitions. This is particularly useful as a way to include a user's special devices. One can "install" the device by simply copying it into a directory.
3. By including the device definition in the library file as an inline cell definition. This is the format used in the supplied `device.lib` file. In early *Xic* releases, this was the only way to define devices.

The native syntax for device definitions, as used in the supplied `device.lib` file, is described below. This is the same CIF-like file format as used in native cell files. The syntax as described applies to native-format device cell files as well as devices inlined into the `device.lib` file. However, for stand-alone cell descriptions, other file formats can be used.

In these cells, there is no physical representation, however an empty physical representation can appear. The default resolution is 100 units per micron (as in CIF and native cell files), however the `(RESOLUTION 1000);` comment can appear, which indicates 1000 units per micron, as in ordinary cells. Each device entry has the following format:

(Symbol *symname*);
5 *property*;
5 ... ;
9 *symname*;
DS 0 1 1;
L SCED;
geometry ...
more layers/geometry ...
DF;
E

The first line is a CIF comment stating the device name, e.g., for a capacitor one might have

```
(Symbol cap);
```

This line signals the beginning of a device definition to the function that automatically updates the device library file after a device is edited (see 8.5), so must appear as shown for that feature to work correctly.

This is followed by property specification lines, which begin with the number '`5`', and a cell name definition, which begins with the number '`9`'. The property lines can occur in any order. Technically, the property lines are optional, however the name line is mandatory. All lines in the symbol specification parts of the file must end with a semicolon (;), except for the symbol termination line "`E`". While the device is being parsed, the ';' is actually taken to be the line terminator, so that logical lines can span several printed lines.

The name line begins with '`9`' in the first column, followed by the symbol name (space separated), and ending with a semicolon (without space). This line actually defines the name of the device, as known to *Xic*. The property lines define the device terminals and other parameters through the property mechanism. Each line begins with '`5`' in the first column, followed by the property number, followed by other data, and finally terminated with a semicolon. Refer to properties description (Appendix D) for information about properties and their syntax. If the device represents a subcircuit macro, the name property must be keyed with the character 'x' or 'X'.

After the property lines comes a CIF define symbol directive:

```
DS 0 1 1;
```

The next line is a directive to use the SCED layer, which is the active layer in the drawing:

```
L SCED;
```

The drawing in the cell should be on this layer to visually match the other elements, however there is no real requirement for this. There are additional layers in the default technology which can be used, typically for highlighting. The geometry used in a device has no electrical significance, i.e., no connectivity, and exists for visual purposes only.

The devices in the supplied `device.lib` file use 100 units per internal "micron" for historical reasons. Be advised that if a `(RESOLUTION 1000);` line appears at the top of the device definition, 1000 units will be assumed for the device. Devices that are edited by *Xic* or added through *Xic* editing will use 1000 units.

After the geometry comes the CIF directive to end symbol definition:

```
DF;
```

The last line of the device definition contains the single character

```
E
```

which indicates the end of the device symbol definition. Note that in this case there is no terminating semicolon.

As an example, here is a sample library entry for a resistor:

```
# resistor
(Symbol res);
5 10 -1 0 0 0 + 0 0 0;
5 10 -1 1 0 -1000 - 0 0 0;
5 11 R 0;
5 15 -100 -100 0 -1 "<v>/<value>";
9 res;
DS 0 1 1;
L SCED;
W 0 0 -1000 0 -750 -100 -700 100 -600 -100 -500 100 -400 -100 -300 0 -250 0 0;
L ETC1;
W 0 -100 -75 -100 -125;
W 0 -125 -100 -75 -100;
DF;
E
```

The property lines (lines beginning with '5') represent two node definitions, a name, and a branch, in that order. The 'W' line (wire) following the SCED layer declaration represents the path used to render the resistor schematic symbol. The other two 'W' lines, following the ETC1 layer declaration, represents a '+' mark used to distinguish the positive end of the resistor, and the target upon which the user clicks to obtain the resistor current, in conjunction with the branch property.

The device library file can be viewed or edited from within *Xic* through the **Open** command. If "device.lib" (or the actual file name) is given in response to the cell-to-edit prompt, a text editing window displaying the file appears. Actually, the current device library file is first copied to the current directory (if it is not already there), and the copy is opened for editing. After saving changes and quitting the text editor, the internal device database is rebuilt from the device library file in the current directory.

Devices from the library can also be edited graphically, and devices added, from within *Xic*. This will be described in the following section.

The terminal device is a special non-physical object used for tying different parts of the circuit together without a wire, and for assigning node names. The library can contain multiple, functionally equivalent terminal devices under various names, each possibly with a different visual style. The name label of a terminal device defaults to the device name, but can be changed by editing the label text once placed. It is important that the name property of the device begin with the character '@'.

In the library, any device that has no name property and exactly one node property will be taken as a ground terminal device. A terminal device will also have exactly one node property, but must have a name property with a name string starting with the '@' character.

See section 7.5 and the subsections that follow for more information about the device menu and the various devices provided in the distributed `device.lib` file.

B.9 Model Library Files

Devices such as transistors require model specification for generation of SPICE simulation input. *Xic* has a mechanism for handling large numbers of device models, while at the same time providing interactive editing capability. Models for devices (`.model` lines) and subcircuits (`.subckt` lines) are read from model library file found along the library search, and from files found in particular subdirectories of the directories in the path.

The first model library file found in the search path is searched for models and subcircuits, as are any files found in subdirectories with a specified name. The default name for model library files is "`model.lib`", however another name can be specified in the technology file with the `ModelLibrary` keyword. The default name for subdirectories to search for device models is "`models`", and this name can be changed in the technology file with the `ModelSubdirs` keyword.

In the search, the current directory is always searched first, whether or not this is actually specified in the search path. Only the first model library file is read, which allows the user to override a system model library file with a custom version. All files found in `models` subdirectories will be searched, unless the directory contains a file names "`.xic_ignore`", in which case the files in the directory will be ignored. The names of the files found in `models` subdirectories are unimportant, but files existing in these directories should contain SPICE models, though it is not an error if no models or subcircuits are found in a file.

As with the device library file, the model library file can be edited using the **Open** command in *Xic*. One simply enters the name (e.g., "`model.lib`") when prompted for the cell name to edit. A text editing window appears. If the file was not found in the current directory, it is copied to the current directory. Otherwise, the file in the current directory is copied to a file with the same name but with a "`.bak`" extension. When editing is complete and changes saved, the model database is rebuilt, using the model library file in the current directory.

The "`models`" subdirectories might be used with large collections of files provided by semiconductor manufacturers. Typically, the package supplied from the manufacturer contains a number of files, each describing a device sold by the manufacturer. In most cases, all that is required to make these models available to *Xic* is to move the files into a `models` subdirectory of a directory in the library path. All of these files found will be added to the database.

The format is that of SPICE, where the first line of each model starts with `.model` (case insensitive), and the text for that model is assumed to extend to the next `.model` or `.subckt` line or end of file.

Subcircuits as well as models are are tabulated. A subcircuit begins with `.subckt`, and ends on a line starting with `.ends`. Models and subcircuits defined within a subcircuit are not accessible as separate library references.

Any line which begins with '`#`' or '`*`' is treated as a comment and ignored.

The text of any of the files in the `models` subdirectories must not change while *Xic* is active. If the text changes after the time that *Xic* caches the file offsets to the models, the model text that *Xic* will extract from the file will very likely be bogus. If the model library file is edited with the **Open** command and saved, all offset tables are updated.

B.9.1 MOS Model Spatial Binning

When *Xic* generates a SPICE netlist, it automatically includes the text of the required models. For MOS devices, i.e., devices keyed by the letter '`m`', a spatial binning model selection scheme is available. This same binning mode is available for MOS devices in *WRspice*. When running *WRspice* from *Xic*, the SPICE text is composed by *Xic*, so the it is usually necessary to resolve the spatial binning within *Xic*.

Complete information is available in the description of the *WRspice* MOS model.

The `L` and `W` parameters values found in the MOS device param property are used to key different models. The model property specifies the basename of the model. The variations are found in the model library under the basename suffixed with "`.1`", "`.2`", etc. Each of these models may contain parameters `LMIN`, `LMAX`, `WMIN`, `WMAX` which specify the parameter window for the model. The model with the window containing the device `L/W` is the one chosen. If a model is missing one of the min/max parameter sets, it will match any value of the parameter.

This page intentionally left blank.

Appendix C

Other File Formats

C.1 Label Font File Format

The font used to render text labels in drawing windows is a vector font for scalability. The character maps have internal defaults, which should be suitable in most cases, however these can be overridden by external definitions from a file. One can dump the current set of character maps to a file with the **Dump Vector Font** button in the font setting panel available in the **Attributes Menu**. Character maps from this file can be modified and placed in a file named "`xic_font`" in the library search path, in which case they will override the internal definitions when producing label text.

The same default character maps are also used by default for the vector font in the **logo** command, for producing physical characters with wire elements. These too can be overridden by definitions from a file. The **Dump Vector Font** button in the setup panel of the **logo** command can be used to dump the current set of character maps to a file. Character maps from this file can be modified and placed in a file named "`xic_logofont`" in the library search path, in which case they will override the internal definitions when producing vector-based characters in the **logo** command.

The generated font file consists of vector specifications for the characters '`!`' through '`~`' in the ASCII chart. The user's file need not contain all characters, missing characters will use the internal default definitions.

The file consists of character specifications of the form described below. The first line of the specification defines the character. This is followed by one or more path vertex lists which define the "strokes" of the character. These are followed by a couple of numerical entries which affect placement. For example, the entry for the default exclamation point (!) appears as:

```
character !
path 4,2 4,7
path 4,9 4,10
width 2
offset 4
```

The coordinate system has its origin in the upper left corner. The size is limited to 256 X 256, but the basic cell size used by the default set is 7 X 14. The y values increase downward, and x values increase to the right. Negative values are not permitted.

Only the first character of the leading keyword is necessary, and this is case insensitive. The first line of the block defines the character. The order of the following lines is unimportant. Each path is a sequence of coordinates which render a part of the character. The `width` is the horizontal space provided for the character, which should include trailing space, typically one column. The `offset` is the column which is placed at the end of the preceding character. Row and column numbering begin with 0.

C.2 Label Flags

Internally, every *Xic* text label object has a set of flags which control presentation and other attributes of the label. The flags are visible in the label specifications in native cell files and default extended CIF files (see B.3). It is also used with script functions (`GetLabelFlags`, `SetLabelFlags`, `Label`, `LabelH`) and the XprpXform pseudo-property (see 10.1.2).

Bits	**Hex**	**Effect**
0-1	0003	text rotation angle
	0000	no rotation
	0001	90 degrees
	0002	180 degrees
	0003	270 degrees
2	0004	mirror Y after rotation
3	0008	mirror X after rotation and mirror Y
4	0010	shift rotations to 45, 135, 225, 315 degrees
5-6	0060	horizontal justification
	0000	left justification
	0020	center horizontal justification
	0040	right justification
	0060	right justification
7-8	0180	vertical justification
	0000	bottom justification
	0080	center vertical justification
	0100	top justification
	0180	top justification
9-10	0600	font selection (unused)
11	0800	unused, reserved
12	1000	show text
13	2000	hide text
14	4000	show only when container is current cell
15	8000	limit number of lines displayed

See the discussion of the `XprpXform` pseudo-property in 10.1.2 for more information on the effects of these flags.

C.3 Help Database Files

The help information is obtained from database files suffixed with `.hlp` found along the help search path. These directories may also contain other files referenced in the help text, such as image files.

In *Xic*, the help search path can be set in the environment with the variable XIC_HLP_PATH, and/or may be set in the technology file (the technology file overrides the environment). These files have a simple format, allowing users to create and modify them. Each help entry is associated with one or more keywords, which should be unique in the database. A warning message will be issued on `stderr` if a name clash is detected. The files are ASCII text, either in DOS or Unix format. Fields are separated by keywords which begin with "`!!`". Although the help system provides rich-text presentation from HTML formatting, entries can be in plain text. A sample plain-text entry has the form:

```
!!KEYWORD
excmd
!!TITLE
Example Command
!!TEXT
    This command exists only in this example.  Note that the
    !!keywords only have effect if they start in the first
    column.  The blank line below is optional.

!!SUBTOPICS
akeyword
anotherkeyword
!!SEEALSO
yetanotherkeyword
```

In this example, the keyword "`excmd`" is used to access the topic, and should be unique among the database entries accessed by the application. The text which appears in the topic (following `!!TEXT`) is shown indented, which is recommended for clarity, but is not required.

In `.hlp` files, outside of `!!TEXT` and `!!HTML` blocks (described below), lines with '`*`' or '`#`' in the first column are ignored, as they are assumed to be comments. Lines that begin in the first column with "`!!`(space)" (space character following two exclamation points) anywhere are also ignored, as comments. Blank lines outside of the `!!TEXT` and `!!HTML` fields are ignored. Leading white space is stripped from all lines read, which can be a problem for maintaining indentation in formatted plain text. To add a space which will not be stripped, one can use the HTML escape "` `".

The following '`!!`' keywords can appear in `.hlp` files. These are recognized only in upper case, and must start in the first text column.

`!!`(space) *anything*
: A line beginning with two exclamation points followed by a space character is ignored.

`!!KEYWORD` *keyword-list*
: This keyword signals the start of a new topic. The *keyword-list* consists of one or more tokens, each of which must be unique among all topics in the database. The words are used to identify the topic, and if more than one is listed, the additional words are equivalent aliases. The *keyword-list* may follow `!!KEYWORD` on the same line, or may be listed in the following line, in which case `!!KEYWORD` should appear alone on the line.

  Punctuation is allowed in keywords, only white space characters can not be used. The '`#`' character has special meaning and should not be part of a keyword name. Also, character sequences that could be confused with a URL or directory path should be avoided. The latter basically prohibits the '`/`' character (and also '`\`' under Windows) from being included in keywords. There are special names starting with '`$`' which are expanded to application-specific internal variables, as described below. To avoid any possibility of a clash, it is probably best to avoid '`$`' in general keywords.

It is often useful to include a meaningful prefix in keywords to ensure uniqueness, for example in *Xic*, all commands have keywords prefixed with "`xic:`".

`!!TITLE` *string*

The `!!TITLE` specifies the title of the topic, and should follow the `!!KEYWORD` specification. The title text can appear on the same line following `!!TITLE`, or on the next line, in which case `!!TITLE` should appear alone in the line. The title is printed at the top of the topic display, and is used in menus of topics.

`!!TEXT`

This line signals the beginning of the topic text, which is expected to be plain text. The keyword is mutually exclusive with the `!!HTML` keyword. The lines following `!!TEXT` up to the next `!!KEYWORD`, `!!SEEALSO`, or `!!SUBTOPICS` line or end of file are read into the display window. The plain text is converted to HTML before being sent to the display in the following manner:

1. The title text is enclosed in `<H1>...</H1>`.
2. Each line of text has a `<BR>` appended.
3. The subtopics and see-alsos are preceded with added `<H3>Subtopics</H3>` and `<H3>References</H3>` lines.
4. The subtopics and see-alsos are converted to links of the form `<A HREF="`*keyword*`">`*title*`</A>` where the *keyword* is the database keyword, and the *title* is the title text for the entry.

Note that the text area can contain HTML tags for various things, such as images. Also note that text formatting is taken from the help file (the `<BR>` breaks lines), and not reformatted at display time. The `!!HTML` line should be used rather than `!!TEXT` if the text requires full HTML formatting.

`!!HTML`

This line signals the beginning of the topic text, which is expected to be HTML-formatted. The keyword is mutually exclusive with the `!!TEXT` keyword. The parser understands all of the standard HTML 3.2 syntax, and a few 4.0 extensions. References are to keywords found in the database and general URLs. Image (`.gif`, etc.) files can be referenced, and are expected to be found along with the `.hlp` files.

`!!IFDEF` *word*

This line can appear in the block of text following `!!TEXT` or `!!HTML`. In conjunction with the `!!ELSE` and `!!ENDIF` directives, it allows for the conditional inclusion of blocks of text in the topic. The *word* is one of the special words defined by the application. Presently, the following words are defined:

`Xic`	Defined when running *Xic* with any feature set.
`XicII`	Defined when running *Xic* with the *XicII* feature set.
`Xiv`	Defined when running *Xic* with the *Xiv* feature set.
`OpenAccess`	Defined when running *Xic* when OpenAccess is available.
`WRspice`	Defined in *WRspice*.
`Windows`	Defined when running under Microsoft Windows.

If *word* is defined, the text up to the next `!!ELSE` or `defined, the text up to the next !!ELSE or !!ENDIF is discarded, and any text following an !!ELSE is included. The constructs can be nested. A word that is not recognized or absent is ‘‘not defined’’. Every !!IFDEF should have a corresponding !!SUBTOPICS lines can appear within the blocks.`

`Example:`

```
!!HTML
   Here is some text.
   You are reading this in Xic.
```

`!!IFNDEF` *word*

`This keyword can appear in the block of text following !!TEXT or !!HTML. It is similar to !!IFDEF but has the reverse logic.`

`!!ELSE`

`This keyword can follow !!IFDEF or !!IFNDEF and defines the start of a block of text to include in the topic if the condition is not satisfied.`

`!!ENDIF`

`This keyword terminates the text blocks to be conditionally included in the topic, using !!IFDEF or !!IFNDEF.`

`!!PROTECT, !!UNPROTECT`

Lines following `!!PROTECT` are ignored until an `!!UNPROTECT` line is seen. This is a means to hide text in the file from the parser.

`!!LATEX` *keyword filename*

This signifies the start of a latex block, and should immediately follow an HTML or TEXT block. It is for use with the **hlp2latex** utility to include LaTex versions of the help text in the file.

`!!INCLUDE` *filename*

The keyword may appear in the text following `!!TEXT` or `!!HTML`. When encountered in the text to be included in the topic, the text of *filename*, which is searched for in the help search path if not an absolute pathname, is added to the displayed text of the current topic. There is no modification of the text from *filename*.

If the filename is a relative path to a subdirectory of one of the directories of a directory in the help search path, the subdirectory is added to the search list. Thus, an HTML document and associated gif files can be placed in a separate subdirectory in the help tree. The HTML document can be referenced from the main help files with a `!!INCLUDE` directive, and there is no need to explicitly change the help search path.

`!!REDIRECT` *keyword target*

This will define *keyword* as an alias for *target*. The *target* can be any input token recognizable by the help system, including URLs, named anchors, and local files. For example:

```
!!REDIRECT nyt http://www.nytimes.com
```

Giving "`!help nyt`" in *Xic* or "`help nyt`" in *WRspice* will bring up a help window containing the New York Times web page.

`!!SEEALSO keyword-list`

This keyword, if used, is expected to be found at the end of the topic text. The *keyword-list* consists of a list of keywords that are expected to be defined by `!!KEYWORD` lines elsewhere in the database. A menu of these items is displayed at the bottom of the topic text, under the heading "References". The keywords specified after `!!SEEALSO` can appear on the same line separated with space, or on multiple lines that follow. If a keyword in the list is not found in the database, the normal action is to ignore the error. The application may provide a debugging mode, whereby unresolved references will produce a warning message.

`!!SUBTOPICS` *keyword-list*
This keyword, if used, is expected to be found at the end of the topic text. This produces a menu of the topics found in the *keyword-list* very similar to `!!SEEALSO`, however under the heading "Subtopics". This can be used in addition to `!!SEEALSO`, the order is unimportant

The following definitions supply header and footer text which will be applied to each page. These should be defined at most once each in the database.

`!!HEADER`
The text that follows, up until the next `!!KEYWORD` or `!!FOOTER`, is saved for inclusion in each page composed from the `!!HTML` lines for database keywords. The header is inserted at the top of the page. There can be only one header defined, and if more than one are found in the help files, the first one read will be used.

In the header text, the literal token `%TITLE%` is replaced with the `!!TITLE` text of the current topic when displayed.

`!!FOOTER`
The text that follows, up until the next `!!KEYWORD` or `!!HEADER`, is saved for inclusion in each page composed from the `!!HTML` lines for database keywords. The footer is inserted at the bottom of the page. There can be only one footer defined, and if more than one are found in the help files, the first one read will be used.

The following keywords implement a means to mark topics that are from imported or supplemental files. For example, in *Xic*, many of the *WRspice* help files are included for reference and to satisfy links in the *Xic* help files. There is a need to mark these pages as applying to the *WRspice* program, otherwise the information could be confusing. In the *Xic* help system, the pages from *WRspice* have a banner just below the header identifying the topic as applying to *WRspice*.

`!!MAINTAG` *tagname*
This keyword should appear once in the database, probably defined along with the header/footer. The *tagname* is an arbitrary short keyword which identifies the database, such as "`Xic`".

`!!TAG` *tagname*
This should be given at the top of each help file in the database. Those files that are part if the main database should have the same *tagname* as was given to `!!MAINTAG`. Files containing supplemental information should have some other *tagname*, e.g., "`WRspice`"

`!!TAGTEXT` *tagname*
This should be given once only in the database, probably where the `!!MAINTAG` is defined. It is followed by HTML text, in the manner of the header and footer. This text will be inserted just below the header in topic pages that come from files with tags that differ from the main tag. For this to happen, both the tag and main tag must have been defined. In the text, the token "`%TAG%`" will be replaced with the actual tag that applies to the topic.

C.3.1 Anchor Text

Clickable references in the HTML text have the usual form:

`<a href="`*something*`">`*highlighted text*`</a>`

Here, "*something*" can be a help database keyword or an ordinary URL.

One can use named anchors in help keywords. This means that the '#' symbol is holy, and should not be used in help keywords. The named anchors can appear in the !!HTML part of the help database entries in the usual HTML way, e.g.

```
!!KEYWORD
somekeyword
...
!!HTML
    ...
    <a name="refname">some text</a>
```

Then, referencing forms like "!help somekeyword#refname" and blather will bring up the "somekeyword" topic, but with "some text" at the top of the help window, rather than the start of the document.

There is an additional capability: '$' expansion. Words found in anchor text that begin with a dollar sign ('$') character may be replaced by either a path related to the program, the value of a variable saved in the program, or the value of an environment variable. The character that immediately follows the word can not be alphanumeric.

This replacement is handled by a callback to the application, but both *Xic* (and its derivatives) and *WRspice* support the following keywords and behavior.

$PROGROOT
: This word is replaced by the full path to the program installation directory, for example "/usr/local/xictools/xic".

$HELP
: This word is replaced by $PROGROOT/help, meaning the same directory as $PROGROOT suffixed with /help.

$EXAMPLES
: This word is replaced by $PROGROOT/examples, as above.

$DOCS
: This word is replaced by $PROGROOT/docs, as above.

$SCRIPTS
: This word is replaced by $PROGROOT/scripts, as above.

If there is no match to these words, the word, without the dollar sign, is checked against the variable database. If a variable is set with the same name, the string value of the variable replaces the word. If there is no match, but the word without the dollar sign matches tne name of an environment variable, the value of the environment variable will replace the word. If there is no match, there is no substitution. Substitutions are evaluated recursively.

If the first character of an anchor URL is '~', the path is tilde expanded. This is done after '$' substitution. Tildes denote a user's home directory: "~/mydir" might expand to "/home/yourhome/mydir", and "~joe/joesdir" might expand to "/home/joe/joesdir", etc.

In *Xic*, one can open input files from anchor text in the HTML viewer. The type of file is recognized by the suffix. These are:

CGX	`.cgx` (.gz may follow)
GDSII	`.gds, .str, .strm, .stream` (.gz may follow)
OASIS	`.oas`
CIF	`.cif`
Xic	`.xic`

The anchor text to open a cell can actually have the following syntax. It can consist of up to three space-separated words.

[*sourcetype*] *sourcename* [*cellname*]

The optional *sourcetype* can be one of the following literal tokens.

`@XIC`
The *sourcename* is the name of a native cell existing either in memory or in the search path for cell files, or the name may contain a path to the file. The *cellname* word is not used.

`@CHD`
The *sourcename* will provide the database name of a cell hierarchy digest. The *cellname* if used provides the name of a cell to open. If not given, the CHD's default cell will be opened.

`@LIB`
The *sourcename* is a path to an *Xic* library file, and the *cellname* is the name of a reference or cell in the library.

`@OA`
The *sourcename* is the name of an OpenAccess library, and the *cellname*, which is required, is the name of a cell in the library.

If no *sourcetype* is given, the file type is determined by the file extension, as listed above. The optional *cellname* can specify the name of a cell to open.

In addition, if the *sourcename* has a `.scr` suffix, it is taken to be a script file, and is executed. Thus, one can execute *Xic* scripts by clicking on an anchor. The referenced script is expected to be found somewhere in the script path, or be defined in the technology file, if a rooted file path is not provided.

Examples:
One can actually load a layout from another machine.

```
Click <a href="http://somewhere/lib/cell.gds">here</a> to view the design.
```

A second argument can specify the cell to open. The quoting is required in this case.

```
Click <a href="/usr/joe/library/joeslayout.gds joescell">here</a> to view Joe's
cell.
```

Unless the native cell happens to have a `.xic` file name extension, one should use the magic word.

```
Click <a href="@XIC mynativecell">here</a> to view my native cell.
```

If the OpenAccess plug-in is loaded, one can access cells from OpenAccess libraries.

```
Click <a href="@OA oalibrary oacell">here</a> to view my OpenAccess cell.
```

Finally, to execute a script when the user clicks on the link:

```
Click <a href="myscript.scr">here</a> to execute myscript.
```

The script `myscript.scr` must exist somewhere in the script path, or be defined in the technology file. When the user clicks on "here", this script will be executed.

In *WRspice*, one can source files from anchor text in the HTML viewer, if the anchor text consists of a file name with a "`.cir`" extension. Thus, if one has a circuit file named "`mycircuit.cir`", and the HTML text in the help window contains a reference like

```
<a html="mycircuit.cir">click here</a>
```

then clicking on the "click here" tag will source `mycircuit.cir` into *WRspice*. Similarly, link references to files with a plot data extension will be loaded into *WRspice* when the anchor is clicked, as if using the **load** command. The known extensions (which of course must match the file type) include "`.raw`" (for rawfiles), "`.csv`" (for comma-separated data (CSV) files), and "`.csdf`" and others for common simulation data format (CSDF) files.

This page intentionally left blank.

Appendix D

Property Specifications

In *Xic*, cells and database objects contain a list of number-string associations called "properties". These are used to store various pieces of information about the object. Some properties a used only by the internals of *Xic* and are not accessible to the user, while other properties can be set by the user to assign certain attributes to an object. The user will encounter properties primarily in electrical mode, as this is the means by which devices are assigned values, models, and other parameters.

The properties that are assigned by *Xic*, and/or have meaning to *Xic* are described if the following sections. Generally, the property numbers 7000 – 7199 are reserved by *Xic*, and property numbers in this range should not be assigned by the user. Also, property numbers in the range 7200 – 7299 correspond to "pseudo-properties" which are used to query or change the parameters of a physical object (see 10.1.2). These values should not be used for assigned properties.

D.1 Physical Mode Property Specifications

This section lists the properties known to *Xic* that may be found in physical cells, instances, or objects. This lists only properties likely to be encountered by the user, there may be additional properties that are not specifically documented used internally by *Xic*. All physical properties known to *Xic* use numbers in the reserved range 7000 – 7199.

text property, number 7012
: This property saves GDSII label parameters ANGLE, MAG, WIDTH, and PTYPE, which are unused by *Xic*. The string consists of a concatenation of keyword/value pairs, using the keywords above (not all need be present). These attributes will be reassigned to the label when a GDSII file is written.

pathtype property, number 7033
: This property is used for physical mode wires of nonzero width which have a non-default path type. The string has the form

  `PATHTYPE` *pathtype*

  where *pathtype* is 0 for flush ends, 1 for rounded ends. The default pathtype is 2 (extended ends). This property is added to wires in native and CIF output if the CifOutExtensions variable has the **wire extension** flag set, and the **wire extension new** flag is **not** set. The wire end style is included in the wire specification in the present default syntax, so **wire extension new** is set by default.

grid property, number 7100

This property is applied to the top-level physical cell when the cell is saved, preserving the current grid setting. This property is used in physical mode only. The property string has the format

`grid` *resol snap*

where *resol* is the number of internal units per snap point, and *snap* is the number of snap points per grid line if positive, or grid lines per snap point if negative.

flags property, number 7105

This property can be be applied to physical cells. The property string can take one of two forms: a hex number, or a space-separated list of string tokens. The tokens and corresponding bits are

Bit	**Keyword**	**Description**
0	`OPAQUE`	When set, the cell is "opaque" with regard to extraction. The cell will look like a black box with terminals.
1	`CONNECTOR`	Not implemented, don't use.
2	`USER0`	User flags, not used by *Xic*. These flags may be useful to the user.
3	`USER1`	

When the ExtractOpaque variable is set, the `OPAQUE` flag is ignored.

refcell property, number 7150

A reference cell is an empty cell with a refcell property, which references a cell hierarchy in another layout file. Reference cells can exist in memory or as a native cell file on disk.

The string for this property consists of space-separated *keyword*=*value* pairs. The known keywords are as follows:

`cellname`
The top-level cell to extract from the referenced hierarchy.

`dbname`
The CHD name in memory. This is never written to a file, it is only used when the cell is in memory.

`filename`
The full path to the referenced layout file.

`bound`
The bounding box, may be used for area filtering, in the form L,B,R,T where the values are floating-point in microns.

`aflags`
Alias flags integer, these set name aliasing modes.

`aprefix`
Cell name change prefix.

`asuffix`
Cell name change suffix.

flatten property, number 7151

During extraction, simple cells that contain only geometry or perhaps all or part of a device can be logically flattened (see 16.4) into their parent cells for extraction purposes. If this property is set in a cell, that cell will always be considered as part of its containing cell by the extraction system.

This is identical to the effect of listing the cell name in the FlattenPrefix variable.

The string for this property is ignored, but is set to "`flatten`" by convention.

nomerge property, number 7152
: The nomerge property applies to physical boxes, polygons, and wires, and is used by the extraction system. If this property is found on any object used to recognize a device body, that device will never be merged with similar devices. This is relevant when merging is enabled for the device during extraction, and one wants to suppress this in individual cases. It prevents both parallel and series merging.

stdvia property, number 7160
: This property is given to standard via sub-masters and instances. The property is recognized by the OpenAccess plug-in providing transparent conversion between OpenAccess and *Xic* standard vias. The syntax is described in 5.8.1.

termorder property, number 7168
: This is set to a space-separated list of group names, and can be applied to physical cells. It will provide the cell connection terminal names and ordering when electrical data are absent. The names must match net name labels (see 16.5) placed in the layout. Names not found are silently ignored.

skipdrc property, number 7178
: This property is applied in output to boxes, polygons, or wires which have the **skip DRC** flag set. It is used to set the **skip DRC** flag in boxes, polygons, and wires as an input file is being read.

labelsize property, number 7180
: This property is added to labels when writing to GDSII, and saves the label width, height and visibility status. The string has the format

    `width` *width* `height` *height* [`show`—`hide`] [`tlev`] [`liml`]

  where *width* and *height* are in internal units.

  The keywords "`show`" or "`hide`" appended to the string store the display state of the label, which can be visible or "hidden", toggled by clicking with button 1 with the **hift** key held. The LabelHiddenMode variable controls the scope of this feature.

  The `tlev` keyword gives the label the property of being invisible in instances of the containing cell, but visible when the cell is viewed as the top-level (current cell).

  The `liml` keyword causes the label to limit the number of lines displayed, when the label text has multiple lines. The maximum line count defaults to 5, and is otherwise given with the `LabelMaxLines` variable.

  The four flags are the same as those accessible with the XprpXform pseudo-property.

This group of properties applies to the OpenAccess interface.

oa_cstmvia property, number 7161
: This property is applied by the translator to *Xic* cells that represent a custom via object from OpenAccess. The format is described in 2.11.

oa_orig property, number 7183
This property is applied transiently when reading cell data into *Xic*. The format is described in 2.11.

The following group of properties implements the Ciranova abutment protocol for parameterized cells. Parameterized cells may use this protocol to automatically merge abutted instances so as to share common features.

ab_class property, number 7185
This is similar to the Ciranova `pycAbutClass` property. The format of the ab_class property is described in 5.5.

ab_rules property, number 7186
This is similar to the Ciranova `pycAbutRules` property. The format of the ab_rules property is described in 5.5.

ab_directs property, number 7187
This is similar to the Ciranova `pycAbutDirections` property. The format of the ab_directs property is described in 5.5.

ab_shapename property, number 7188
This is similar to the Ciranova `pycShapeName` property. The format of the ab_shapename property is described in 5.5.

ab_pinsize property, number 7189
This is similar to the Ciranova `pycPinSize` property. The format of the ab_pinsize property is described in 5.5.

ab_inst property, number 7190
The format of the ab_inst property is described in 5.5.

ab_prior property, number 7191
The format of the ab_prior property is described in 5.5.

ab_copy property, number 7192 The format of the ab_copy property is described in 5.5.

The following property is required to implement the Ciranova protocol for stretch handles in parameterized cells. A stretch handle is a display element that can be dragged with the mouse, which initiates a change of a cell property and appropriate remastering.

grip property, number 7195
The format of the grip property is described in 5.4.

The remaining properties support parameterized cells (pcells). The super-master pcell contains a script reference, default parameter values, and (optionally) parameter constraint strings. When the super-master is instantiated, the script is executed producing a sub-master under a modified name, plus an instance of the sub-master. The instance contains the name of the super-master and a copy of the instantiation parameters.

pc_name property, number 7197
This property is assigned by *Xic* to pcell sub-masters and their instances. It provides the name of the pcell from which the sub-master or instance was derived.

pc_params property, number 7198
This property is assigned by the user to pcells, and contains the default parameter set. It will be assigned by *Xic* to sub-masters and instances, and contains the parameter set that was used to create the sub-master. See 5.1.3 for a complete description.

pc_script property, number 7199
This property is assigned by the user to a pcell, and appears only in the super-master. It contains the script, or a path to a script, which is executed when the pcell is instantiated. See 5.1.3 for a complete description.

D.2 User-Specified Electrical Property Specifications

The properties described in this section provide user-specified information to device and subcircuit instances, and to device and cell definitions. In many cases, the property applied to a device definition will supply a default for a similar property created in the new instance when the device is instantiated. The instance property can be subsequently modified by the user.

The name property described in the next section, plus the devref, model, value and param properties discussed below, translate into fields of device definition lines when generating SPICE output, and in order to set these properties proficiently, the user must have familiarity with the SPICE syntax.

The strings for these properties may contain special escape sequences indicating hypertext references or other characteristics. These are described in D.4.

model property, number 1
The model property appears in device instances and defines a device model to be included in the SPICE line for the device. This property is normally assigned to the device instance with the **Property Editor** from the **Edit Menu**, but a default model can be supplied by including this property in the device definition in the device library file.

`5 1` *model_name*`;`

The *model_name* is arbitrary, but a corresponding entry should exist in a model library file.

value property, number 2
The value property supplies a string to be used in the device line in SPICE output for the device "value". The property is normally applied to device instances with the **Property Editor**, but can appear in the device definition in the device library file to assign a default value for the device.

`5 2` *value*`;`

The *value* is a string which may, for example, represent a floating point number specifying the component value, e.g., in ohms for a resistor. In general, any string can appear, and it may include hypertext references. A complex string would be necessary for a voltage source with functional dependence, for example.

param property, number 3
The param property specifies the part of the device SPICE line which provides an initial condition or other data not included in a model or value string. The property is normally applied to device and subcircuit instances with the **Property Editor**, or to cells with the **Cell Property Editor** command. When applied to cells or subcircuit instances, the property is used to provide parameter definitions for SPICE (see the description of the `.subckt` line in the *WRspice* manual). This can

also appear in the device definition in the device library file to provide a default. If given to a cell, instances of the cell will inherit the property, which can then be changed from within *Xic* on a per-instance basis. For device instances, this property specifies any parameter, such as device area, which is provided in the device line after the model. This manifestation was referred to as the initial condition ("initc") property in previous documentation.

5 3 *string*;

The *string* will be appended to the device line when a SPICE file is created. It can contain initial condition data or other parameters significant to the device, which are syntactically expected to the right of the model or value.

The parameter definitions in a param property string have the form

name1=*value1 name2*=*value2* ...

There may be white space around the '=' character. The *name* tokens are parameter names, which are alphanumeric words starting with an alpha character. The *value* token can not be empty, and must be a single text token. This means that if the *value* string contains white space, it must be single or double quoted. Be aware that the interpretation of single quoted (`'word'`) and double quoted (`"word"`) differs fundamentally. Double quoting implies a manifest string type. The string will be assigned verbatim to the parameter, which will be of string type. No further processing will be done. Single quoting implies an expression which reduces to a number when evaluated. If a *value* is not quoted, it will be evaluated as an expression if necessary, otherwise it will be taken as a numeric value. Generally, parameter assignment failures are silently ignored.

other property, number 4

The other property is a catch-all device property that is not used by *Xic* and does not appear in SPICE output. There can be arbitrarily many other properties specified for a device, unlike the model, value, and param properties which can appear at most once. The other property can be used for storage of alternate values for the model, value, and param properties. It is applied to device instances with the **Property Editor**. Although it can be used in device definitions in the device library file, there seems to be no reason for doing so.

5 4 *string*;

nophys property, number 5

When the nophys property is applied to an electrical device or subcircuit, that device or subcircuit is assumed to have no physical implementation and is ignored in the algorithm that associates electrical and physical devices and subcircuits. A device or subcircuit with this property has no dual in the physical layout, and its terminals will never be placed in the physical layout, where they would otherwise be visible with the **Show Terms** command. Devices and subcircuits with this property will be ignored in LVS testing.

In order to actually simulate a circuit that has been extracted from the physical layout, it is necessary to add sources and perhaps other devices, which have no counterparts in the physical layout. In general, this will cause LVS errors in subsequent LVS runs. The nophys property can be added to the additional devices to avoid these errors.

By "ignoring" these devices, the device terminals are considered as open circuits. However, there are times when it would be useful to consider these devices as shorted. For example, suppose that one wishes to include parasitic series inductance in a resistor during simulation. However, this inductance would cause LVS to fail, since the series inductor added to the schematic has no explicit physical counterpart.

It is possible to configure the nophys property to indicate that when the electrical netlist is generated for use by the extraction system, the corresponding devices will be forced such that all terminals connect to the same net, i.e., the terminals are effectively shorted together. Thus, the inductor in the example above, if given this property, would disappear properly during LVS.

The numerical value of the property is 5. The property string is either "`nophys`" or "`shorted`". The latter indicates that the shorting feature is to be used. *Xic* will always set the property string to one of these values. Devices inherit this property from cell definitions in the device library file. The format is

`5 5 nophys;` or
`5 5 shorted;`

Devices with the nophys property applied will be rendered using a different color than other devices.

`virtual` property, number 6

When the virtual property is applied to an electrical subcircuit, the subcircuit will not be included in netlist output. This means that in SPICE output, the corresponding "`.subckt`" block of lines will be absent. However, calls to this subcircuit, if any, will be included, and must be resolved through text from a `.include` line or by some other means.

This is a method for including "foreign" subcircuits within the *Xic*/*WRspice* framework.

The numerical value of the property is 6. The property string is "`virtual`". *Xic* will always set the property string to this value. This property applies only to electrical cell definitions (subcircuits). The format is

`5 5 virtual;`

`flatten` property, number 7

This can be applied to electrical masters and instances. The state is active if the instance has the property and the master does not, or the instance does not have the property and the master does. If active, the schematic will be logically flattened into its parent before association in LVS.

`range` property, number 8

The range property can be applied to device (other than terminal devices) and subcircuit instances. The property contains two non-negative integers, which define a range of values between the start and end integers inclusive, stepping by one. When applied to an instance, the instance becomes *vectorized*, with the range providing the subscripts for the individual scalar instances. Scalar contact terminals become vectors, and vectors become bundles. Use of vector instances can simplify some schematics with repeated circuit blocks. More information about vector instances and the rules for connecting to them can be found in 4.2.9.

The property number is 8, and the property string consists of two non-negative integers, the starting and ending values of the subscripting range. The property applies only to non-terminal device instances and subcell instances.

`macro` property, number 20

The macro property is **no longer in use**, having been replaced by the `macro` flag which is associated with the name property. However, it is still recognized and performs its intended function when encountered. By default, it will not be generated in output, thus there is a potential compatibility issue with *Xic* release 4.3.5 and earlier. The new variable WriteMacroProps can be set before generating output to include macro properties, thus providing backwards compatibility.

While reading input, if a macro property is read, a window appears reminding the user to set WriteMacroProps if backwards compatibility is needed. The message can be avoided by either of the following:

1. Save the design to a new file, it will not be backwards compatible, and will have no macro properties.
2. Set the WriteMacroProps variable in a startup script. This suppresses the message, and backwards-compatible files will be produced.

It is no longer possible to (conveniently) create macro properties in *Xic*, for example with the **Cell Property Editor**.

The `macro` flag (or property) applies to device master cells. When present, its only effect is that in SPICE output, an '`X`' is prepended to the device name in instantiation lines of the device. Thus, SPICE will treat the device instance as a subcircuit call. These instances must have a model property giving a name that will match a `.subckt` definition somewhere, likely from a PDK device model file.

This accounts for devices that are likely the electrical part of parameterized cells, that implement nonlinear behavior through a network of controlled sources expressed as a subcircuit in the SPICE model definitions file. MOS capacitors and poly resistors are devices that are frequently modeled this way.

If the `macro` flag is set and the name *prefix* already begins with `X` or `x`, the device is taken as a macro, meaning that *Xic* will not output a subcircuit definition for the cell, and a model property will provide the name of a subcircuit definition expected to be found in the model library or elsewhere.

`devref` property, number 21

This property maps text that appears in a SPICE device call after the node list but ahead of the model or value. The purpose is to provide the name of a reference device for current-controlled sources (CCCS and CCVS), and the current-controlled switch (CSW). This property can be applied to device instances only, and is supplied by the user typically with the **Property Editor**.

The property supports hypertext, and the reference name should be added as a hypertext reference, so that the correct device is referenced if the name should change. That is, when editing the property string on the prompt line, click on the device to reference. The device name will be entered in the line using colored text, indicating a hypertext entry. Unlike plain test, the hypertext entry will still be correct if the referenced device name changes.

There is no default, and a missing property will produce a syntax error in a generated SPICE file.

D.3 *Xic*-Managed Electrical Property Specifications

The properties that are set by *Xic* in electrical mode are described below. The electrical property values use the integers 1–21. The values 22–30 are reserved for future use.

bnode property, number 9

The bnode property identifies the location of a "bus connector" which is used to specify multiple connections to a device or subcircuit. It may appear in subcircuit cell definitions and instance references.

5 9 *index beg_range end_range x y*;

The *index* is a non-negative integer index which serves to link the bus node to an existing node with the same index. The *beg_range* and *end_range* are non-negative integers which set the indexing of the bus. The bus bit indices range from *beg_range* through *end_range*. Note that the numbers can be ascending or descending. The "bit" for *beg_range* is connected to the node with the given *index*. The "bit" for *end_range* is connected to the node with index equal to *index* + `abs(`*beg_range* - *end_range*`)`. If no node property has an equal index value, then that "bit" is simply open.

For cells, the *elecX* and *elecY* each have the general form

schemX[`:`*symbX*[`,`*symbX* ...]]

and similar for the Y values. This represents a single X,Y contact location in the schematic, and an arbitrary number of contact locations in the schematic symbol. The schematic value is separated from the symbolic values by a colon. The symbolic values are separated from each other by commas. If the 3.2 format is being written due to the Out32nodes variable being set, at most one number will appear following the colon, the first that would otherwise be listed if there are multiple contact points.

If there is no symbolic representation, or the terminal location has not been set in the symbolic view, the *elecX* and *elecY* each consist of a single number. In the more general case, both terms should supply the same number of integers.

In cell instances, there is no colon delimiter, and the general form is simply a comma-separated list of numbers. This is all identical to the coordinate specification for a node property.

node property, number 10

The node property defines a circuit connection point. It appears as a property of wires, device and subcircuit instances, and cells. Its string is a bit different in the three cases.

Wire property

`5 10` *circuit_node*

Any text that follows the form shown above is ignored. The *circuit_node* is the node number in the current cell of the net containing the wire. All wires that participate in connectivity, i.e., on the `SCED` layer and any layer with the `WireActive` technology file keyword applied, should have a node property.

Instance property

`5 10` *circuit_node index elecX elecY* [*name*]

A subcircuit or device instance will have one node property per circuit connection. The *circuit_node* is the node number of the connection in the current cell. The *index* is the terminal ordering parameter. Each node property of an instance will have a unique *index*. The indices form a compact run starting with 0.

The *elecX* and *elecY*, are integers, or comma-separated lists of integers. Both terms specify the same number of integers. Taken as ordered X,Y values, these provide the "hot spots" where connection to the terminal can be made. *Xic* allows an arbitrary number of hot spots per node. If the Out32nodes variable is set, which forces output compatible with earlier *Xic* releases, the *elecX* and *elecY* will each consist of a single value, the first in the list that would otherwise be output if there are multiple contact points.

Internally, there are 1000 integer counts per "micron", and hot spots must appear on a 1 "micron" (1000 unit) grid. In a cell file, scaling may be applied. In particular, the default for CIF and native cell files is 100 units per micron, but this is usually changed to 1000 units with the `RESOLUTION 1000` directive. Anyway, if a user is for some reason writing a node property string by hand, the hot spot locations must be chosen appropriately, arbitrary locations do not work.

Anything that follows is actually ignored by the reader. The terminal's name is printed in output since it might be of interest to humans.

Cell property

`5 10` *circuit_node index elecX elecY* [`0x`*flagstype name phyX physY layer_name*];

Cell property, old 3.2 syntax

`5 10` *circuit_node index elecX elecY* [*name phyX physY flags layer_name type_name*];

The `node` properties applied to a cell make it possible for the cell to be instantiated and used as a subcircuit (or device) in another cell.

The property string parser can recognize and read the old release 3.2 string format for compatibility. When writing output in any file format, if the variable `Out32nodes` is set, the old string format will be generated. This will, however, strip out multiple contact points if any have been defined, as this is not supported by the older format, which allows exactly one contact per node. The variable tracks the **Use back-compatible format (warning! data loss)** check box in the **Export Control** panel from the **Export Cell Data** button in the **Convert Menu**.

The *circuit_node* is the node number in the current cell where contact is to be made. In a device cell that has no internal nodes, this will be -1. The *index* is an ordering parameter as discussed above. Index zero is the reference node. When the device or subcircuit is placed in a schematic, the location of the reference node corresponds to where the user clicks.

The *elecX* and *elecY* each have the general form

schemX[`:`*symbX*[`,`*symbX* ...]]

and similar for the Y values. This represents a single X,Y contact location in the schematic, and an arbitrary number of contact locations in the schematic symbol. The schematic value is separated from the symbolic values by a colon. The symbolic values are separated from each other by commas. If the 3.2 format is being written due to the `Out32nodes` variable being set, at most one number will appear following the colon, the first that would otherwise be listed if there are multiple contact points.

If there is no symbolic representation, or the terminal location has not been set in the symbolic view, the *elecX* and *elecY* each consist of a single number. In the more general case, both terms should supply the same number of integers.

The remaining tokens are optional. The *flagstype* is a hex integer that if present **must** be prefixed with "`0x`" or "`0X`". The least significant byte contains a value that specifies a terminal type. This is a numerical equivalent of the optional *type_name* which appears in the old 3.2 format syntax. The values and keywords are listed below. *Xic* does not presently use this.

value	**keyword**
0	`input` (default)
1	`output`
2	`inout`
3	`tristate`
4	`clock`
5	`outclock`
6	`supply`
7	`outsupply`
8	`ground`

The remainder of the word may contain any of the following flag bits.

`0x100` (BYNAME)
: The terminal will associate to a wire net by name, there will be no connectivity due to placement location in the schematic.

`0x200` (VIRTUAL)
: The terminal is “virtual” meaning that there is no wire vertex or subcircuit or device contact at the terminal’s location in the schematic. This is irrelevant if the BYNAME flag is set.

`0x400` (FIXED)
: It set, *Xic* will not move the corresponding physical terminal location in the layout. This indicates that the location has been “locked” by the user.

`0x800` (INVIS)
: The terminal will be invisible in the schematic, except when the **subct** command from the side menu, used for terminal editing, is active. The corresponding terminal in the layout will show normally.

The *name* is the terminal name. This is either a short name provided by the user, or if not provided a default name will be created by *Xic*. The name is unique among the cell’s terminals.

If the cell has a physical counterpart, the remaining arguments have significance. In particular, if the cell has no physical counterpart, or the node has no physical counterpart, the remaining parameters should not appear. The lack of physical coordinates informs the reader that this terminal has no physical counterpart. The coordinates should be set to zero in device cells that have a physical implementation.

If the node has a corresponding physical implementation in the layout, the *physX* and *physY* will be given. When the property is being read, the presence of these numbers indicates that internal setup to link to the layout is required. If both numbers don’t appear, the node will exist in the schematic only. This is appropriate for devices that don’t have a physical implementation, such as voltage sources, or for cases like the phase node of a Josephson junction, or in the case where there is no layout. In output, if the physical association exists. the two numbers give the corresponding point in the layout. These will be nonzero if the corresponding location has been identified, either by running extraction, or if the location was set by hand.

It is not necessarily true that all nodes of the device either have or don’t have the optional parameters. The phase node of a Josephson junction device, for example, does not have a physical counterpart. The other two nodes do have physical implementations, since these are the physical connection points. A side-effect is that in SPICE files extracted from physical data only the two nodes will appear in the device instantiation lines. This is acceptable to *WRspice*, since the phase node is optional. If a device has no nodes with the optional parameters given, then it can never have a physical counterpart. The nophys property should also be given in that case. This is true for devices like voltage sources that have no physical implementation.

If the physical location is valid, a *layer_name* will be provided. This is the name of a physical layer which has the Conductor attribute, and an object on this layer touches or is under the physical location.

In the old 3.2 format, the flags values are the following:

`0x2` (VIRTUAL)
: The terminal is “virtual” meaning that there is no wire vertex or subcircuit or device contact at the terminal’s location in the schematic. This is irrelevant if the BYNAME flag is set.

`0x4` (FIXED)
: It set, *Xic* will not move the corresponding physical terminal location in the layout. This indicates that the location has been “locked” by the user.

In the old format, instead of a numerical type code, an optional *type_name* can be given. This is one of the keywords from the table shown earlier.

`name` property, number 11

The `name` property gives the device an identifying prefix or name. If a name has been assigned to the device with the **Property Editor** panel or equivalent, that name will be used in SPICE output. Otherwise, the name prefix is suffixed with a unique integer generated by *Xic* to form the name. SPICE expects that the first character of the name match the convention for the device, for example, resistors use R, capacitors C, etc. (see the SPICE documentation).

Cell property, `macro` flag set:

`5 11` *prefix* `macro`

Cell property, *prefix* starts with `X` or `x`:

`5 11` *prefix* `0 subckt`

Cell property, otherwise:

`5 11` *prefix*

Cell instance property:

`5 11` *prefix.assigned_name devnum* [`subckt` [*physX physY*]];

The *prefix* is the default name prefix, and should conform to the SPICE conventions. The *assigned_name*, if present, will be used in actual spice output. The *assigned_name* should **not** be present in device definitions, it is used in cell files for device instances to which a name has been assigned with the **Property Editor**. The *prefix* can start with any character, but is intended to have significance to SPICE. The character '`@`' is reserved for the terminal device. The *assigned_name* can be any contiguous string. The *devnum* is an index assigned by *Xic* to the device, and is used when forming the default device name. When reading, this value is ignored.

The `name` property for cells contains an internal `macro` flag, which replaces the `macro` property in 4.3.6 and later. This flag will be set if any of the following apply:

1. Exactly two words are given, i.e., a single word follows the *prefix*, which can be anything.
2. The word "`macro`" appears in a third word, following an integer.
3. A `macro` property is found.

This syntax is backwards compatible with release 4.3.5 and earlier.

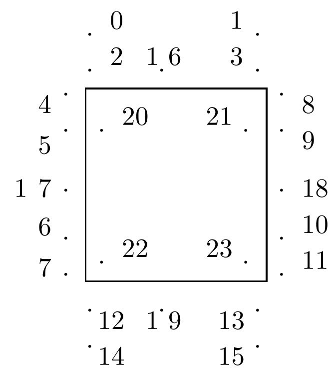
When the `macro` flag is set, its only effect is that in SPICE output, an '`X`' is prepended to the device name in device instantiation lines. Thus, SPICE will treat the device instance as a subcircuit call. These instances must have a `model` property that will match a `.subckt` definition somewhere, likely from a PDK device model file.

This accounts for devices that are likely the electrical part of parameterized cells, that implement nonlinear behavior through a network of controlled sources expressed as a subcircuit in the SPICE model definitions file. MOS capacitors and poly resistors are devices that are frequently modeled this way.

If the `macro` flag is set and the *prefix* already begins with `X` or `x`, the device is taken as a macro, meaning that *Xic* will not output a subcircuit definition for the cell, and a `model` property will provide the name of a subcircuit definition expected to be found in the model library or elsewhere.

The name property for instances is printed as shown. The "`subckt`" appears if *prefix* starts with "`X`" or "`x`" and `macro` is **not** set in the master. Coordinates additionally appear if a physical label

Figure D.1: Locations and justification for character position codes around the device bounding box.



was placed (in extraction). This is where the physical subcircuit instance label is located. All but the *prefix* and *assigned_name* (if any) are ignored by the reader, but can be seen printed in native cell files (for example).

Xic generates the internal device or subcircuit index, used as part of the default device or subcircuit name, according to the position of the upper-left corner of the bounding box of the object. The numbering starts with zero, and increases for positions with smaller Y value, or with larger X value for devices with the same Y coordinate. Each device and subcircuit type has its own numbering.

labloc property, number 12

The name, model, value, param, and devref property values are normally displayed on-screen near the device body. This is a device property for setting the default locations of the property labels when shown on-screen. If this property does not appear, the internal default locations are used. This property allows more control over label placement, on a per-device basis. This property should only be used in devices in the device library file. Presently, the property can only be added with a text editor by editing the property strings in the device library file.

`5 12` *pname code* `[` *pname code* `] ... ;`

The *pname* is one of the literal tokens "**name**", "**model**", "**value**", "**param**", and "**devref**". For backward compatibility, "**initc**" is accepted as an alias for "**param**". The *code* is an integer, -1 – 23. If the *code* is -1, the default placement is used. If code is 0 – 23, the placement and justification are as shown in the figure: The '.' position implies the justification. Horizontally, all are left or right justified except for 16 and 19 which are centered. Similarly, vertical justification is bottom or top except for 17 and 18 which are center justified.

The default locations are shwon in the table below. The locations differ when the height of the placement bounding box is less than or greater than the width.

Property	**height > width**	**height <= width**
name	5	2
model	8	13
value	8	13
param	11	14
devref	18	12

oldmut property, number 13

This property is used for compatibility with the mutual inductors used in the schematic files produced by the Jspice3 program. The format should not be used, and is not documented.

mut property, number 14
This property appears with the properties of cells containing mutual inductors, and is not copied to instantiations of the cell. Mutual inductors do not appear as devices in the device library file, rather, they are implemented with this property. Mutual properties are generated by selecting the "mut" device from the device menu, with one property assigned for each mutual inductor pair in the circuit.

`5 14` *num name1 num1 name2 num2 coeff* `[`*name*`];`

This property appears only in the list for cell definitions, and not for instances. It defines a mutual inductor pair within the cell. The *num* is the index of the mutual inductor pair, used in forming the default specification to SPICE: "`K`*num*". However, if the *name* appears (supplied in *Xic* by using the label editor on a mutual inductor label), the SPICE specification will use *name* (without *num*). The *name1, num1, name2, num2* are the prefixes and assigned numbers of the inductors in the mutual inductor pair. The *coeff* is a string which represents the coupling factor as given to SPICE.

branch property, number 15
The branch property is used to define a "hot spot" that when clicked on yields a device parameter, such as device current, which can be used in plots. In SPICE, voltage sources and inductors have internal storage for current values present by default. Other device parameters may require additional computational or storage overhead. If the branch property is given in the device definition in the device library file, it is added to instantiated devices by *Xic*.

`5 15` *x y dx dy* `[`*string*`];`

The *x* and *y* values specify the hot spot where the branch current can be accessed by clicking. The next two numbers specify the assumed direction of current flow. They are interpreted as a unit vector directed outward from the origin along the +/− x or y axes. Thus,

direction	*dx*	*dy*
+y	0	1
−y	0	−1
+x	1	0
−x	−1	0

are the options. The *string* will be expanded and added to the token list in the prompt line when the branch is selected for plotting.

When the hot spot is clicked on, an expression will be produced which after expansion is added to the input line in the **plot** command. The *string* token can contain the following literal tokens, which will be replaced with the appropriate values during expansion:

`<v>`	Voltage across the device
`<value>`	The "value" property
`<name>`	The device name

Anything else in the *string* will be copied literally. If the string is absent, the expression will be "`<name>#branch`".

Here are some examples. for a resistor, the string is

```
<v>/<value>
```

to return the current. Similarly for a capacitor,

```
<value>*deriv(<v>).
```

Thus the current will be computed using the *WRspice* `deriv` function. For an inductor or voltage source, no string is required, as the default

```
<name>#branch
```

is appropriate. For a current source, one can use

```
@<name>[c].
```

This works through the *WRspice* `@`*device*`[`*param*`]` mechanism, however the vector must be saved, most conveniently by setting the LibSave global property for the device (see B.8).

labref property, number 16

The labrf property is applied by *Xic* to labels that are associated with device properties or wire nodes. The property is not used in the device library file.

`5 16` *name num property*`; 5 16` x y `10;`

This property applies only to labels, and indicates that the label is to be bound to a given property of a certain device or mutual inductor, or wire.

The first form applies to a label for an instance or mutual inductor property. Bound labels automatically reflect changes in the underlying property string, and may be used to set the string using the label editing function in *Xic*. The *name* and *num* are the device prefix and assigned number of the device to which the label is bound. The *property* is the property number of the bound property. If the label is assigned to a mutual inductor pair, the *name* is '`K`'.

The second form applies to labels that have been attached to a wire, and are used to contribute a name for the net containing the wire. The x and y specify the coordinates of a vertex of the wire. The `10` is the value of the node property.

mutlrf property, number 17

This property is assigned to inductor instances which are referenced for use in mutual inductor pairs. One such property exists per reference. It is not used in device library files.

```
5 17 mutual;
```

This property applies only to inductors that are referenced as one of a mutual inductor pair. There can be several such properties if the inductor is associated with multiple mutual inductor pairs.

symbolic property, number 18

The symbolic property is a property applied to cells which have a symbolic view associated. It does not appear in device library files, as all devices are essentially symbolic. It is not inherited by instances.

`5 18 0/1` *geometry_spec*`;`

The third field is nonzero if the symbolic mode is active, and 0 if symbolic mode is inactive. The *geometry_spec* is a string of separated CIF primitives for the symbolic representation, which can include L, B, P, W, and 94 (label) directives. Each record (CIF primitive) is terminated by a colon (**not** a semicolon!) which must be immediately followed by an end-of-line character. Colons that are not at the end of a text line will **not** terminate a record.

The symbolic property may be applied to subcircuit instances, in which case it will negate the effect of a symbolic property found in the instance master cell. In this utilization, the property is named **nosymb**. The flag and *geometry_spec* are ignored and need not be provided. A subcircuit instance with this property would (in all cases) be displayed as expanded. Thus, it is possible in *Xic* to have different instances of the same subcircuit cell master display symbolically and expanded within the same containing cell.

nodemap property, number 19

The nodemap property is applied to the electrical cell definition and is not inherited by instances. The nodemap property provides a mapping between internally generated node numbers and assigned textual names.

`5 19 0/1` *name x y name x y ...*;

The third token can be 0 or 1, but is unused. In releases prior to 3.1.5, a 0 value would disable the node map. In current releases, node mapping is always enabled.

The remainder of the line consists of triples containing an assigned name and a coordinate pair. The coordinates correspond to a device or subcircuit terminal connected to the assigned node, and serve as the reference to that node. See 7.11 for more information on node mapping.

The "global" properties are added to the electrical top level cell of a hierarchy when being saved. They save plot points and other information in the file, to use as defaults when the file is subsequently loaded for editing.

run property, number 7101

The run property string specifies the default analysis command entered when the **run** button is pressed which initiates a simulation.

plot property, number 7102

The Plot property is used only in electrical mode. The string represents the plot points used in the **plot** command, in the format of arguments to the *WRspice* `plot` command.

iplot property, number 7103

The Iplot property is used in electrical mode. The string specifies the points to plot when using the **iplot** button, in the format of arguments to the *WRspice* `plot` command.

The strings for the plot and iplot properties may contain special escape sequences indicating hypertext references or other characteristics. These are described in D.4.

In SPICE, Each line of a given device type begins with the device name, set by the name property. This is followed by the device nodes, corresponding to the order of enumeration in the *device_node* of the node properties. This is followed by text from the devref property, which is intended to provide a reference device name for current-controlled sources and the current-controlled switch. It is not used generally. This is followed by the value or model property (these are really just two different names for the same text field). This is followed by the text of the param property.

The device name, if not assigned by the user with the **Property Editor** command, and nodes are assigned by *Xic* so as to be unique.

The line looks like:

name n1 ... nLast devref value/model parameter_string

The *name* is either the user assigned name, or the device prefix with a unique numerical suffix created by *Xic* if no name was assigned. The nodes can be numbers or text tokens, in accordance with the current node name mapping (see 7.11). The remaining properties are read verbatim from the specifications, with hypertext references expanded.

Hypertext references are generated when assigning properties by clicking on devices or other features in the drawing. Since *Xic* assigns device names and nodes, if one needs to reference a specific device or node, a hypertext reference provides a link which is independent of the assigned values, which can change.

When applied to subcircuit cells and instances, the **param** property provides support for subcircuit parameterization, which is available in *WRspice* and some other simulators.

Here is a brief description of how to use parameterization. Suppose that you are editing a cell that contains a resistor, and you wish to parameterize the resistance value. Give the resistor a value property consisting of some word, say "`rshunt`". Using the **Cell Properties Editor**, give the cell a param property something like "`rshunt=2.5`". This will give the resistor a default value of 2.5 ohms. Editing another cell, place two instances of the previous cell. Using the **Property Editor** give one of the instances a param property of "`rshunt=5`". A label will appear containing this text. The other instance will not have a similar label. The resistor in the labeled subcircuit will have value 5, set by the param property applied to the instance. The other instance will have resistance value 2.5, as set by the param property applied to the master, which serves as the default value.

D.4 Special Escapes

In property and label strings, there is a special encoding used to indicate certain attributes, such as hypertext references. These are in the form:

(||*something*||)

The following forms are recognized by *Xic*

(||sc||)
: This sequence is simply converted to a semicolon ('`;`') when the string is internalized. In CIF, semicolons can not be included in label or property strings, as the character is reserved for line termination. *Xic* will convert semicolons in property strings and labels to this form when creating a CIF or native file.

(||text||)
: This token may appear at the beginning of a label string, and indicates that the string is in long text format (see 7.9.5). These labels do not appear on-screen (the characters "`[text]`" appear instead), but the full string can be accessed with the label editor. Thus, large blocks of text can be saved as properties or `spicetext` labels without crowding the screen.

(||x:y *type*||)
: This sequence indicates a hypertext reference, and can appear anywhere in a property or label string, in electrical data only. Hypertext references are generated when assigning properties by clicking on other devices in the drawing. Since *Xic* by default internally assigns device names and nodes, if one needs to reference a specific device or node, a hypertext reference provides a link which is independent of the assigned values, which can change. The x,y is a coordinate, in internal units, giving a location for the reference. This is generally the point where the user clicked to

create the reference. The space-separated integer that follows gives the type of the reference, and is one of:

1	node reference
2	branch reference
4	device name reference
8	subcircuit name reference

In the case of a node reference, the coordinate must be over a connection point, or along a wire. For a branch, the coordinate must be over a branch reference point of a device. For a device name reference, the coordinate must be in or on the bounding box of a device. For a subcircuit name reference, the coordinate must be in or on the bounding box of a subcircuit.

When the string is used, the hypertext reference is resolved, and the actual text replaces the hypertext reference in the string.

Appendix E

Xic Variables

Xic maintains an internal list of keyword/value associations. Although this list can be used for general purposes, there are a number of special keywords, or "variables", whose value will affect *Xic* operation. Variables are set with the **!set** command, and can be unset with the **!unset** command. The script functions `Set`, `Unset`, `SetExpand`, and `Get` also provide an interface to this database. Variables can be set from the technology file, and a number of the buttons in menus and various pop-ups really do nothing more than control the state of one of these variables.

Any variable name can be set with the **!set** command. The variables and constructs that have meaning to *Xic* are summarized in the table below. These are described more fully in the sections that follow.

Special Constructs	
`!set`	List variables currently set
`!set ?`	List these variables
`@`*devname.property*	Set device property
Startup	
DatabaseResolution	Set internal units
NetNamesCaseSens	Net names are case-sensitive
Subscripting	Set net name subscripting character
DrfDebug	Report undefined layer attribute names
Paths and Directories	
Path	Design data file search path
LibPath	Startup file and library search path
HelpPath	Help file search path
ScriptPath	Script file search path
NoReadExclusive	Don't move stripped path to front of search path
AddToBack	Add stripped path to back of search path
DocsDir	Directory containing release documentation
ProgramRoot	Set to the program's installation directory
TeePrompt	Copy messages to given filename or "stdout"
General Visual	
MouseWheel	Set mouse wheel rate parameters
ListPageEntries	Maximum entries per page in list pop-ups
NoInstnameLables	Don't use instance names in unexpanded instances

NoLocalImage	Don't compose images locally
NoPixmapStore	Don't use screen backing memory
NoDisplayCache	Don't use multi-object rendering for boxes
LowerWinOffset	Pixel spacing of pop-up windows above prompt line
PhysGridOrigin	Set the origin of the grid displayed in physical mode
ScreenCoords	Show window pixel coordinates
PixelDelta	Cursor selection proximity is screen pixels
NoPhysRedraw	When set, don't redraw physical windows after layer visibility change
NoToTop	Don't move obscured windows to top
'!' Commands	
Shell	Path to shell used for external commands
OpenAccess Interface	
OaLibraryPath	Set location for hidden libraries
OaDefLibrary	Default library name
OaDefTechLibrary	Default technology attachment library
OaDefLayoutView	Default layout view name
OaDefSchematicView	Default schematic view name
OaDefSymbolView	Default symbol view name
OaDefDevPropView	Default device property view name
OaDmSystem	Set design management system
OaDumpCdfFiles	Dump CDF data to a file
OaUseOnly	Restrict to physical/electrical data
Parameterized Cells	
PCellAbutMode	Control pcell auto-abutment
PCellHideGrips	Hide stretch handles if set
PCellGripInstSize	Instance size threshold for stretch handles
PCellKeepSubMasters	Include pcell sub-masters in file output
PCellListSubMasters	Include pcell sub-masters in modified cells list
PCellScriptPath	Search path for pcell scripts
PCellShowAllWarnings	Show warnings during pcell evaluation
Standard Vias	
ViaKeepSubMasters	Include standard via sub-masters in file output
ViaListSubMasters	Include standard via sub-masters in modified cells list
Scripts	
LogIsLog10	The log function returns base-10 when set
Selections	
MarkInstanceOrigin	Show origin of selected instances
MarkObjectCentroid	Show centroids of selected physical objects
SelectTime	Set delay (msec) to activate move
NoAltSelection	Use legacy click-selection logic
MaxBlinkingObjects	Maximum number of objects shown blinking
Side Menu Commands	
MasterMenuLength	Maximum masters in **Cell Placement Control** menu
DevMenuStyle	Set presentation style of device menu
LabelDefHeight	Default text label height in microns
LabelMaxLen	Max length of displayed label string
LabelMaxLines	Max lines of displayed label string
LabelHiddenMode	Set scope for hidden labels

LogoEndStyle	End style for logos: 0 flush, 1 round, 2 extend
LogoPathWidth	Path width for logos, 1 – 5
LogoAltFont	Specify alternate font for logos
LogoPrettyFont	Name of system font to use for logos
LogoPixelSize	Specify the "pixel" size for logos
LogoToFile	Create subcell for logos
NoConstrainRound	No DRC constraints creating round objects
RoundFlashSides	Number of sides to use in physical round objects
ElecRoundFlashSides SpotSize	Number of sides to use in electrical round objects Set mask resolution
SPICE Interface	
SpiceListAll	Include unconnected devices in Spice output
SpiceAlias	Device key aliases for Spice output
SpiceHost	Name of *WRspice* server
SpiceHostDisplay	X display string to use on remote host
SpiceInclude	Add include file to SPICE netlist
SpiceProg	Path name of *WRspice* executable, supersedes below
SpiceExecDir	Directory containing *WRspice* executable
SpiceExecName	Name of *WRspice* executable
SpiceSubcCatchar	Character used by *WRspice* in subcircuit expansion
SpiceSubcCatmode	Mode for *WRspice* subcircuit expansion
CheckSolitary	Report unconnected terminals in netlist
NoSpiceTools	Do not show *WRspice* toolbar
File Menu – Printing	
NoAskFileAction	Don't ask before file actions in File Selection pop-up
DefaultPrintCmd	Default print command (printer name in Windows)
NoDriverLabels	Don't use driver text for hard copy labels
RmTempFileMinutes	Set up temporary file removal
Cell Menu Commands	
ContextDarkPcnt	Control illumination of context in **Push** command
Editing General	
AskSaveNative	Prompt to save modified native cell when editing new cell
Constrain45	Constrain polygon and wire angles to 45-degree multiples
NoMergeObjects	Suppress merging new boxes, polygons
NoMergePolys	Clip/merge boxes only when merging
NoFixRot45	Don't "fix" vertex locations after non-Manhattan rotation
Edit/Modify Menu Commands	
UndoListLength	Number of operations saved in the undo list
MaxGhostDepth	Maximum subcell expansion depth in ghosting
MaxGhostObjects	Maximum number of objects shown in ghosting
NoWireWidthMag	Don't change the width of magnified wires
CrCellOverwrite	Allow Create Cell to overwrite memory cells
LayerChangeMode	Specify layer change during move/copy
JoinMaxPolyVerts	Upper bound of vertices in polygons from join (def. 600)
JoinMaxPolyGroup	Limit number trapezoids per poly in join (def. 300)
JoinMaxPolyQueue	Limit number trapezoids to form polys in join (def. 1000)
JoinBreakClean	Manhattan split polygons with too many vertices
JoinSplitWires	Include wires in join/split operations

PartitionSize	Partition grid size in microns for layer operations
Threads	Number of helper threads to employ
View Menu Commands	
InfoInternal	Use internal coordinates in info windows
PeekSleepMsec	Per-layer delay in peek command, milliseconds
LockMode	Don't allow physical/electrical mode change
XSectNoAutoY	Disable cross-section automatic Y scaling
XSectYScale	Set cross-section Y scale factor
Attributes Menu Commands	
TechNoPrintPatMap	Use hex format for stipple maps when writing tech file
TechPrintDefaults	Set printing of default values in tech file update
BoxLineStyle	Line style mask for highlighting box
EraseBehindProps	Erase behind phys properties in props command
PhysPropTextSize	Pixel text height used in props command
EraseBehindTerms	Erase behind physical mode terminals marks
TermTextSize	Pixel height of text used in terminal marks
TermMarkSize	Pixel width of cross used for terminal marks
ShowDots	Control electrical connections display
FullWinCursor	Enable full-window cursor
CellThreshold	Min size in pixels of displayed subcell, integer >= 0
GridNoCoarseOnly	Don't show coarse grid without fine grid
GridThreshold	Minimum visible grid spacing pixels
Convert Menu – General	
ChdFailOnUnresolved	Halt CHD operation if unresolved cell
ChdCmpThreshold	Set CHD compression block size threshold
MultiMapOk	Allow non-1–1 mapping of *Xic* layers and GDSII layer/datatypes
NoPopUpLog	Don't pop up log file if warnings or errors
UnknownGdsLayerBase	Base number for generated GDSII layers
UnknownGdsDatatype	Datatype for generated GDSII layers
NoStrictCellnames	Allow white space in cell names
NoFlattenStdVias	Keep standard via instances when flattening
NoFlattenPCells	Keep parameterized cell instances when flattening
NoFlattenLabels	Ignore labels in subcells when flattening
NoReadLabels	Ignore text labels when reading physical cell data
KeepBadArchive	Don't delete failed conversion output archive file
Convert Menu – Input and ASCII Output	
ChdLoadTopOnly	Load requested cell from CHD only, create reference
ChdRandomGzip	Use random-access table for gzipped files
AutoRename	Automatically change clashing cell names when reading
NoCreateLayer	Don't create new layers when reading
NoMapDatatypes	New layers take all datatypes in GDSII read
NoAskOverwrite	Suppress prompting for overwrite instructions
NoOverwritePhys	Don't overwrite phys memory cells when reading
NoOverwriteElec	Don't overwrite elec memory cells when reading
NoOverwriteLibCells	Don't overwrite library cells when reading
NoCheckEmpties	Skip checking for empty cells while reading
NoPolyCheck	Skip polygon reentrancy tests when reading
DupCheckMode	Check for duplicate items when reading

EvalOaPCells	Attempt to create sub-master for OpenAccess pcell instances
NoEvalNativePCells	Don't attempt to create sub-master for native pcell instances
MergeInput	Merge boxes and coincident objects when reading
LayerList	Layer list for conversion input filtering
UseLayerList	How to use layer list, skip or use only
LayerAlias	List of name=alias pairs
UseLayerAlias	Map layers using layer alias list
InToLower	Map lower case cell names to upper in archive read
InToUpper	Map upper case cell names to lower in archive read
InUseAlias	Use alias file when reading archive
InCellNamePrefix	Cell name translation prefix for archive read
InCellNameSuffix	Cell name translation suffix for archive read
CifLayerMode	CIF layer resolution method, 0–2
OasReadNoChecksum	Ignore checksum in OASIS input file
OasPrintNoWrap	Use one line per record in OASIS ASCII output
OasPrintOffset	Add file offsets to OASIS ASCII output
Convert Menu – Output	
StripForExport	Strip all format extensions from output file
WriteMacroProps	Include deprecated macro properties in output
KeepLibMasters	Write library cells when creating archive file
SkipInvisible	Do not write invisible layers to output
NoCompressContext	Don't compress instance lists in archive context
RefCellAutoRename	Use auto-rename when writing reference cell data
UseCellTab	Enable use of the cell override table in CHD access
SkipOverrideCells	Skip cells in override table in CHD access
OutAllCells	Output all cells in symbol table, not only current
Out32nodes	Use old 3.2 node property syntax in output
OutToLower	Map lower case cell names to upper in archive write
OutToUpper	Map upper case cell names to lower in archive write
OutUseAlias	Use alias file when writing archive
OutCellNamePrefix	Cell name translation prefix for archive write
OutCellNameSuffix	Cell name translation suffix for archive write
CifOutStyle	CIF output dialect and extensions specifier
CifOutExtensions	CIF output extension flags
CifAddBBox	Add bounding box comment to objects in CIF output
GdsOutLevel	GDSII release level conformance code (0–2)
GdsMunit	Modify M-UNITS value in GDSII output file
GdsTruncateLongStrings	Cut strings too long for record
NoGdsMapOk	Ignore unmapped layers in GDSII/OASIS output
OasWriteCompressed	Compress records in OASIS output
OasWriteNameTab	Use string table referencing in OASIS output
OasWriteRep	Try to combine similar objects in OASIS output
OasWriteChecksum	Compute and add checksum to OASIS output
OasWriteNoTrapezoids	Don't convert polys to trapezoids
OasWriteWireToBox	Convert wires to boxes when possible
OasWriteRndWireToPoly	Convert rounded-end wires to polygons
OasWriteNoGCDcheck	Don't look for common divisors in repetitions
OasWriteUseFastSort	Use faster but less effective sorting

OasWritePrptyMask	Don't write certain properties
Custom Property Filtering	
PhysPrpFltCell	Physical cell property filter string
PhysPrpFltInst	Physical instance property filter string
PhysPrpFltObj	Physical object property filter string
ElecPrpFltCell	Electrical cell property filter string
ElecPrpFltInst	Electrical instance property filter string
ElecPrpFltObj	Electrical object property filter string
Design Rule Checking	
Drc	Enable interactive rule checking
DrcNoPopup	Suppress violation reporting pop-up
DrcLevel	Set violation reporting level
DrcMaxErrors	Quit testing when this many violations found
DrcInterMaxObjs	Maximum number of objects to test interactively
DrcInterMaxTime	Maximum milliseconds for interactive test
DrcInterMaxErrors	Maximum violation count for interactive test
DrcInterSkipInst	Skip expensive instance check in interactive test
DrcChdName	Name of CHD for batch test
DrcChdCell	Name of top cell in CHD to test
DrcLayerList	List of layer names for filtering
DrcUseLayerList	Use only or skip layers in list
DrcRuleList	List of rule names for filtering
DrcUseRuleList	Use only or skip rule in list
DrcPartitionSize	Partition grid size in microns
Extraction Tech	
AntennaTotal	Default input for **!antenna** command
Db3ZoidLimit	Trapezoid limit for the 3-D database
LayerReorderMode	Default layer sequencing option
NoPlanarize	When set, no layers are assumed planarizing
SubstrateEps	Relative dielectric constant of substrate
SubstrateThickness	Assumed thickness of substrate in microns
Extraction General	
ExtractOpaque	Ignore the OPAQUE flag in extraction
FlattenPrefix	Cell name prefix to flatten in extraction
GlobalExclude	Layer expression to exclude objects during extraction
GroundPlaneGlobal	Ground all pieces of clear-field ground plane
GroundPlaneMulti	Handle nets in dark-field ground plane
GroundPlaneMethod	Set ground plane inversion method 0–2
KeepSortedDevs	Include devices with terminals shorted
MaxAssocLoops	Maximum loop count for association
MaxAssocIters	Maximum iteration count for association
NoMeasure	Suppress measuring parameters of devices
UseMeasurePrpty	Read and update cached measurement results property
NoReadMeasurePrpty	Don't read cached measurement results from property
NoMergeParallel	Never merge parallel devices
NoMergeSeries	Never merge series devices
NoMergeShorted	Never merge devices with all terminals shorted
IgnoreNetLabels	Ignore labels found in nets

UpdateNetLabels	Create or update net labels after association
FindOldTermLabels	Search for old-style "term labels"
MergeMatchingNamed	Merge nets with the same logical net name
MergePhysContacts	Merge contacts for split-net handling
NoPermute	Skip permutation search in association
PinLayer	Name of layer for net labels
PinPurpose	Name of purpose for net labels
RLSolverDelta	Overriding grid spacing for resistance/inductance extraction
RLSolverTryTile	Attempt to use tiling grid for resistance/inductance extraction
RLSolverGridPoints	Grid points per device when not tiling
RLSolverMaxPoints	Maximum grid points per device when tiling
SubcPermutationFix	Apply post-association permutation fix
VerbosePromptline	Print info on prompt line during extraction
ViaCheckBtwnSubs	Check connectivity between subcircuit nets by via
ViaSearchDepth	Cell hierarchy depth to search for vias
ViaConvex	Assume all vias are convex polygons
Extract Menu Commands	
QpathGroundPlane	**"Quick" Path**, use of inverted ground plane, 0–2
QpathUseConductor	**"Quick" Path**, allow Conductor objects in net
EnetNet	Print net, **enet** command
EnetSpice	Do include SPICE listing, **enet** command
EnetBottomUp	Use leaf-to-root ordering in electrical netlist
PnetNet	Print extracted net list, **pnet** command
PnetDevs	Print extracted device list, **pnet** command
PnetSpice	Print extracted SPICE list, **pnet** command
PnetBottomUp	Use leaf-to-root ordering in physical netlist
PnetShowGeometry	Include wire geometry in netlist file, **pnet** command
PnetIncludeWireCap	Include routing caps in SPICE netlist, **pnet** command
PnetListAll	List ignored and flattened subcells, **pnet** command
PnetNoLabels	No net names from labels in **pnet** command output
PnetVerbose	Print more information in **pnet** command output
SourceAllDevs	Update internal-named devices in **sourc** command
SourceCreate	Create devices in **sourc** command even if not empty
SourceClear	Clear cell before updating with **sourc** command
SourceGndDevName	Name of ground device used with **sourc** command
SourceTermDevName	Name of terminal device used with **sourc** command
NoExsetAllDevs	Don't use internal-named devices in **exset** command
NoExsetCreate	Don't create devices in **exset** command
ExsetClear	Clear cells before updating in **exset** command
ExsetIncludeWireCap	Include routing capacitance in **exset** command
ExsetNoLabels	No net names from labels in **exset** command output
LvsFailNoConnect	Force LVS failure if unconnected physical instance
PathFileVias	Include vias in wire net files
Capacitance Extraction Interface	
FcArgs	Capacitance extractor command line arguments
FcForeg	Run capacitance extractor in foreground if set
FcLayerName	Capacitance extractor masking layer name
FcMonitor	Capacitance extractor output appears in console window if set

FcPlaneTarget	Refined element count target
FcPath	Path to capacitance extractor executable
FcPlaneBloat	Capacitance extractor substrate bloat dimension
FcUnits	Capacitance extractor file units: m, cm, mm, um, in, mils
Inductance/Resistance Extraction Interface	
FhArgs	*FastHenry* command line arguments
FhDefaults	Text for `.DEFAULT` line in *FastHenry* input
FhDefNhinc	Default for `nhinc` in *FastHenry* input
FhDefRh	Default for `rh` in *FastHenry* input
FhForeg	*FastHenry* run in foreground if set
FhFreq	*FastHenry* frequency specification
FhLayerName	*FastHenry* interface masking layer name
FhManhGridCnt	Manhattanization grid cell count
FhMonitor	*FastHenry* output appears in console window if set
FhOverride	Override `nhinc`, `rh` in *FastHenry* input
FhPath	Path to *FastHenry* executable
FhUnits	*FastHenry* file units: m, cm, mm, um, in, mils
FhUseFilament	Use *FastHenry* filaments
FhVolElMin	*FastHenry* volume element minimum size factor
FhVolElTarget	*FastHenry* volume element count target
FhVolEnable	Enable segment refinement.
Help System	
HelpDefaultTopic	Suppress or set the default help topic
HelpMultiWin	Use separate windows for help references

E.1 Special Constructs

These are special **!set** variables and constructs which have significance to *Xic*.

(no arg)

Pop up a list of the currently set variables. Variables in this list (with the exception of the path variables) can be removed with the **!unset** command.

?

Pop up a list of the variables that have meaning to *Xic*.

`@`*devname.property*

Set the *property* on device *devname* to *value*. This construct enables device properties to be added to devices via the command line. The first character of the *name* token must be '`@`', followed by the name of the device, a period, and the name of the property to set. Valid property names are "`name`", "`model`", "`value`", "`param`", "`other`", and "`nophys`". For backward compatibility, "`initc`" is recognized as an alias for "`param`". An unrecognized property name will be saved as an "other" property.

Examples:

`!set @L2.value 2ph`

sets the value of L2 to 2ph.

`!set @Moutput.param L=2`
sets the length parameter of mosfet `Moutput`.

The *devname* field can be the name of a mutual inductor, in which case the valid properties are "`name`" and "`value`".

E.2 Startup

The following variables control fundamental behavior of the *Xic* program. These must be specified before reading design or technology data. Unlike all other variables, these can be set only from the `.xicinit` file, which is read before the technology file, or the technology file. These can not be set or unset in a `.xicstart` file, which is read after the technology file, unless no technology file is read. They can not be set by any other means.

The `Set` script function can be used in the initialization files to set this variable. In the technology file, the `!set` command should be used, and this must appear at the top of the file, before layer definitions.

DatabaseResolution
Value: string: "`1000`", "`2000`", "`5000`, or "`10000`".
By default, *Xic* uses an internal resolution of 1000 units per micron. In releases prior to 3.0.12, this was internally hard-coded. As the dimensions used in integrated circuits continue to shrink, an option for higher resolution was added through use of the DatabaseResolution variable.

The internal resolution can be set with this variable, to one of the listed choices. If unset, 1000 units is used. This resolution applies only to physical data, electrical resolution is fixed at 1000.

Superficially, changing the internal resolution has only subtle effects from the user's vantage point. Some of these are:

1. If not 1000, four digits following the decimal point are used when printing coordinates in microns, in many places in *Xic*. Otherwise, only three digits are used.
2. The ultimate zoom-in and grid spacing sizes are smaller for higher resolutions.
3. The size of "infinity", the maximum accessible size for the design, becomes smaller as resolution is increased, since coordinates are stored internally as 32-bit integers. For 1000 units, the field width is approximately 2 meters, which decreases to 20 centimeters at 10000 units. This should still be plenty for most purposes.
4. Layout files produced by *Xic* will use the internal resolution, so that no accuracy is lost.

NetNamesCaseSens
Value: boolean.
By default, net names are case-insensitive in *Xic*, and saved internally as upper-case. If this boolean variable is set, net names are taken as case-sensitive. This impacts lookup of nets by name and comparison of net names for identification and matching purposes, as used in the electrical schematic and extraction system.

Subscripting
Value: string.
In *Xic*, net name and cell instance indexing can employ angle, square, or curly brackets, as in the forms `mynet<1>`, `mynet[1]`, and `mynet{1}`. These forms are equivalent and can be freely mixed in *Xic* input.

However, on occasion *Xic* will create a vector name for output. The default is to use angle brackets, but this can be changed by setting this variable. The variable must be set to a word starting with one of the letters `a`, `s`, or `c`, case insensitive (the "word" can be just the letter). Only the first letter is significant. The letters signify angle, square, or curly brackets.

DrfDebug

Value: boolean.
This obscure flag applies when using the `ReadDRF`, `ReadCdsTech` and `ReadCniTech` technology file directives. If this variable is set, non-serious warnings encountered when reading these files will be printed. One such warning is generated by use in the Virtuoso or Ciranova technology file of color, stipple, or packet names that have not been defined in the display resourse file (DRF). Since there are defaults, these unresolved name references are not a serious problem.

At least one commercial process design kit had lots of these issues, and reporting these as warnings on every *Xic* startup became irritating, particularly since it is not something that the typical user can fix, or want to bother with fixing. Thus, these not-really-errors are ignored by default, but if the user desires then setting this variable will make any such errors visible.

This variable must be set before the files are read. Setting this variable at the top of the *Xic* technology file with the `!set` construct is a convenient way.

E.3 Paths and Directories

These variables set the search paths (see 2.6) and document directory used in *Xic*. These have counterpart environment variables (see 2.5). The search paths can also be set from the technology file.

If not set by any means, internal defaults are used for the search paths and document directory. Under Windows, the default is set to point to the actual installation location subdirectories when necessary. Under Unix/Linux, the XT_PREFIX environment variable should be set to the installation location prefix that effectively replaces "`/usr/local`".

Below, *PREFIX* is obtained from the Windows Registry database under Windows, which is defined when the program is installed. Under Unix/Linux, *PREFIX* is obtained from the XT_PREFIX environment variable. In both cases, the default value for *PREFIX*, if another definition is not found, is "`/usr/local`".

Path

Value: path string, can't be unset.
This variable contains the design data search path. It is always defined, and can not be unset. This path is used to find native cell, archive, and library files.

If not set by any means, a default path is used.
Default: "(.)"

LibPath

Value: path string, can't be unset.
This variable contains the startup library search path. It is always defined, and can not be unset. The library path is used to find the technology file, device and model libraries, and other initialization files.

Unlike other search paths, the current directory is *always* searched first, whether or not this is indicated in the search path string. If not set by any means, a default library path is used.
Default: "(. *PREFIX*`/xictools/xic/startup`)"

HelpPath

Value: path string, can't be unset.
This variable contains the help search path. It is always defined, and can not be unset. This path is used to find files that contain information for the help system.

If not set by any means, a default help path is used.
Default: "(*PREFIX*`/xictools/xic/help`)"

ScriptPath

Value: path string, can't be unset.
This variable contains the script search path. It is always defined, and can not be unset. This path is used to find script and menu files that will appear in the **User Menu**.

If not set by any means, a default script path is used.
Default: "(*PREFIX*`/xictools/xic/scripts`)"

The treatment of any path which is given with a native cell to open in the **Open** command can be altered with the next two variables.

NoReadExclusive

Value: boolean.
When a native cell name with a path is opened, the path is stripped from the cell name. If the path is not already in the search path, it is added. Ordinarily, the path is put in front of the search path for the duration of the read, so that subcells will be opened from the same directory. If this variable is set, the path is not necessarily moved to the front of the search path.

AddToBack

Value: boolean.
A path stripped from a given cell name in the **Open** command is added to the back of the search path, rather than the front.

The behavior is described below for the various permutations:

NoReadExclusive unset
AddToBack unset
(default behavior)

The directory is added to the front of the search path during the read. The "." element of the path, if it exists, is moved to the front after the read.

NoReadExclusive unset
AddToBack set

The directory is added to the front of the search path during the read. The "." element of the path, if it exists, is moved to the front, and the directory is moved to the end after the read.

NoReadExclusive set
AddToBack unset

If the directory exists in the path, nothing is changed, otherwise the directory is added to the front. After the read, the "." entry, if it exists, is moved to the front.

NoReadExclusive set
AddToBack set

If the directory exists in the path, nothing is changed, otherwise the directory is added to the end.

DocsDir

Value: path to directory.
The given directory is searched for the release notes, for the **Release Notes** command in the **Help Menu**.

If not set by any means, a default document directory is used.
Default: "*PREFIX*`/xictools/xic/docs`"

ProgramRoot

Value: string.
This variable is set by the program to the installation location assumed by the program at program start-up. For example, for *Xic* installed in the standard location, the variable will contain the string "`/usr/local/xictools/xic`". This variable is not used by *Xic*, but is available in scripts so that the user can query the value when needing to access files in the installation location. Note that the user can set or clear this variable arbitrarily.

TeePrompt

Value: path to file.
When set, the prompt line messages are copied to the given file. If a file name is not given, or when the variable is unset, redirection stops. The value string can be "`stderr`" or "`stdout`" to redirect output to the terminal window instead of a file.

E.4 General Visual

The following **!set** variables affect general visual attributes of *Xic*.

MouseWheel

Value: two floating-point numbers.
This variable controls the per-click increments for mouse wheel panning and zooming of drawing windows. Without a key held, the mouse wheel scrolls drawing windows up/down. If **Shift** is held, scrolling is right/left. If **Ctrl** is held (overrides **Shift**), the mouse wheel zooms out or in.

The string provided to this variable consists of two space-separated floating-point numbers, each in the range of 0 – 0.5. The first is the pan factor, the second is the zoom factor. The default is `0.1 0.1`. Larger numbers increase the effect per mouse wheel click. If either number is set to 0, that effect (pan or zoom) is suppressed. Thus, to turn off mouse wheel support in drawing windows, give "`0 0`".

ListPageEntries

Value: integer 100–50000
This sets the number of entries that appear per page in the pop-ups that list cells. If the number of cells to be listed exceeds this number, a page menu will become visible in the listing panel. Each page will contain at most this number of entries. Only the entries for the currently selected page will be visible. If this variable is not set, the default value is 5000.

NoInstnameLabels

Value: boolean.

Starting in release 4.3.3, the label used in physical display windows for unexpanded cell instances is the instance name, which consists of the master cell name followed by a colon separator and a unique integer index. When this variable is set, the label shows the master cell name only, the same as in earlier *Xic* releases.

NoLocalImage

Value: boolean.

In *Xic* generation 3, a "local image" may be used to compose images for screen rendering. The display image is composed in local memory, and flushed to the screen when drawing is complete. When using X-Windows, this provides much faster rendering of complex displays, particularly when running remotely over a network, than the standard method of server-side image manipulation as used exclusively in previous *Xic* releases.

The local image method is not used under Windows, since it provides no benefit in the Windows architecture. It is also not used if the hierarchy being shown is not complex, i.e., contains few subcells and objects, as the conventional drawing mode is quicker in this case.

If this variable is set, the local image feature is disabled, and rendering is always performed by server-side functions. This is for debugging, it is not likely that the user will need to set this variable.

NoPixmapStore

Value: boolean.

In normal operation, the screen refreshes are buffered through an in-core pixel map. The geometry is rendered in the map, and when finished the map is copied to the screen. This is generally faster than drawing directly to the screen. When this variable is set, all drawing is direct to the screen. This is intended only for debugging purposes.

NoDisplayCache

Value: boolean.

In normal operation, boxes are cached during rendering, and displayed with a multiple object rendering call. This should be faster than rendering the boxes individually. When this variable is set, the caching is disabled. This is intended only for debugging purposes.

LowerWinOffset

Value: integer -16 to 16.

For windows that are automatically placed just above the prompt line, giving this variable a positive value will position these windows toward the top of the screen by that many pixels. This is useful when using "plasma" displays (such as Mac or KDE), where the shadow falls on the prompt line, which can be distracting. It might also be helpful if the window positioning is incorrect, which might occur with some window managers. This variable tracks the state of the **Pixels between pop-ups and prompt line** entry area in the **General** page of the **Window Attributes** panel from the **Attributes Menu**.

PhysGridOrigin

Value: two floating-point numbers.

This will set the origin of the displayed grid in physical-mode windows. The value consists of two floating-point numbers, which are taken as the x and y grid origin location in microns. This applies only to the displayed grid, and specifically not to the grid/snap used when creating or locating objects.

When an offset is active, the word `"PhGridOffs"` will be displayed in the status area.

ScreenCoords

Value: boolean.

When set, the coordinate readout area will display the position of the mouse pointer in the current drawing window in the window's pixel coordinates. This is for development/debugging purposes and is not likely to be useful to the user, and in fact may cause trouble if used while editing.

PixelDelta

Value: integer (default 3).

This variable determines how close, in screen pixels, a user must click to a feature for *Xic* to recognize this as clicking "on" that feature. The value should likely be set larger than the default for very high-resolution screens, or for inaccurate pointing devices, or for users with less than the sharpest eyesight.

NoPhysRedraw

Value: boolean.

When set, physical windows will not be redrawn after a layer visibility change in the layer table. This is traditional behavior of earlier *Xic* releases, which assumed that screen redraws would take some time and the user would prefer to force a redraw when desired.

NoToTop

Value: boolean.

By default, most if not all *Xic* sub-windows will automatically rise to the top if completely covered by the *Xic* main window. This includes plot windows from WRspice running under control of *Xic* (however most window managers don't support this). If this variable is set, the action will be disabled. This will apply to plot windows from *WRspice* that is started after the variable is set.

Some (probably most) window managers will do this automatically for sub-windows, in which case setting this variable will have no effect on the *Xic* sub-windows, but would still affect *WRspice* plot windows if the window manager supports this. The only window manager I know of that supports this is Exceed 2008, because it is old. The protocol is deemed a security risk and has been disabled in modern window managers for some time.

E.5 Keyboard '!' Commands

The **!set** variables below affect the '!' commands available from the keyboard. Commands of this form that are not recognized as internal commands are assumed to be operating system commands, and are executed in a separate window under a command shell.

Shell

Value: string.

This variable can be set to the name of a command interpreter which will be used for the '!' and !*shellcmd* inputs. The interpreter will be instantiated in its own window. If not given, the shell program used will be taken from the SHELL environment variable, and if this variable is not found the default is "`/bin/sh`". *WRspice* users can set the shell to "`wrspice`" for quick access to the full user interface of that program.

Under Microsoft Windows, the value must be a full path name to the shell executable, and the COMSPEC environment variable is also consulted for the default shell, after the SHELL variable.

E.6 OpenAccess Interface

The following **!set** variables affect the OpenAccess interface. These variables have no effect unless the OpenAccess plug-in is loaded.

OaLibraryPath
Value: string.
This can specify a path to a directory, which will be searched if a library can not be found. When opening a library, and the library is not found in the `lib.defs` (or `cds.lib`) file, the system will look for the library as a subdirectory of the directory path specified in this variable, if any. This allows use of OpenAccess libraries that are hidden from other tools.

OaDefLibrary
Value: string.
This can be set to the name of a library in the OpenAccess `lib.defs` (or `cds.lib`) file, or a subdirectory of the OaLibraryPath if any. This will be used as the default library in certain commands, if no other library is given. Presently, the **!oabrand**, **!oasave**, and **!oaload** commands use this.

OaDefTechLibrary
Value: string.
This can be set to the name of a library in the OpenAccess `lib.defs` (or `cds.lib`) file, or a subdirectory of the OaLibraryPath if any. When a library is created, it will attach the technology database associated with the library name found in this variable, if set. If the named library has an attached technology, the same attachment will be applied to the new library. Otherwise, the new library will attach the local technology database of the named library.

OaDefLayoutView
Value: string.
This sets the view name assumed for physical data in OpenAccess. When not set, the default is "`layout`". This variable tracks an entry area in the **OpenAccess Defaults** panel.

OaDefSchematicView
Value: string.
This sets the view name assumed for schematic data in OpenAccess. When not set, the default is "`schematic`". This variable tracks an entry area in the **OpenAccess Defaults** panel.

OaDefSymbolView
Value: string.
This sets the view name assumed for symbol data in OpenAccess. When not set, the default is "`symbol`". This variable tracks an entry area in the **OpenAccess Defaults** panel.

OaDefDevPropView
Value: string.
This provides a default name for a simulator-specific view from which device properties are obtained. Often these peoperties are in a format intended for a specific simulator. If not set, the default is "`HspiceD`", which assumes the Hspice simulator, to which the *WRspice* simulator has compatibility. This variable tracks an entry area in the **OpenAccess Defaults** panel.

OaDmSystem
Value: string.
If this variable is set to a string starting with '`t`' or '`T`', OpenAccess will be set to use the Turbo design management system. Otherwise, OpenAccess will use the default FileSys system. Compatibility with Cadence seems to require use of the FileSys system. The Turbo system is claimed to

have higher performance. The format of information stored on disk is very different in the two approaches. Supposedly, this should be invisible to the OpenAccess user.

OaDefTechLibrary

Value: boolean.

If this variable is set when a parameterized cell is opened in OpenAccess, the CDF data for the cell will be dumped to a file in the current directory. The file name is the cell name with a "`.cdf`" extension. This is for development/debugging.

OaUseOnly

Value: string.

This variable can be used to limit data imported from and exported to OpenAccess to physical only or electrical only. The variable tracks the state of the **Data to use from OA** radio group in the **OpenAccess Libraries** panel.

If set to "1", or to any text starting with 'p' or 'P', only physical layout data will be read from or written to OpenAccess. If set to "2", or to any text starting with 'e' or 'E', only electrical data (schematic and symbol) will be read or written. If not set, or set to anything else, both physical and electrical data will be read or written.

The restriction applies to conversion to and from OpenAccess, by any method in *Xic*.

One useful observation is that one can import a schematic from Virtuoso even if no provision has been made to export pcells. Unless the Express PCell feature is enabled in Virtuoso, conversion of Skill-based pcells will fail as they can not be evaluated outside of the Cadence environment. The Express PCell feature makes available a cache of pre-built sub-masters that can be exported. Without this, attempting to import physical data will produce a lot of errors, which can be avoided if only a schematic is needed by importing electrical data only.

E.7 Parameterized Cells

The following **!set** variables affect parameterized cell (pcell) capabilities. Most of these track elements of the **PCell Control** panel obtained from the **PCell Control** button in the **Edit Menu**.

PCellAbutMode

Value: integer 0–2, default 1.

Xic provides an internal implementation of the Ciranova auto-abutment protocol (see 5.5). This variable sets the value of the otherPinsOnNet parameter mentioned in the protocol description. How the pcell uses this variable is up to the pcell author, there is really no *a-priori* interpretation, it is an integer of value 0, 1, or 2.

The Ciranova Nmos2 example pcell interprets the value to have the following meanings. This is likely to be used in other pcells as well.

0	Auto-abutment is disabled.
1	Abutment takes place with no contact between the gates.
2	Abutment takes place with a M1 contact between the gates.

This variable tracks the **Auto-abutment mode** selection menu in the **PCell Control** panel.

PCellHideGrips

Value: boolean.

Xic implements the Ciranova stretch handle protocol (see 5.4), and by default stretch handles are visible in selected, expanded cell instances, and when editing the sub-master. If this variable is set, all stretch handles will be invisible and disabled.

This variable tracks the state of the **Hide and disable stretch handles** check box in the **PCell Control** panel.

PCellGripInstSize
Value: integer 0–1000.
Stretch handles are not shown and inactive if the instance rendering size is too small. This is to avoid triggering a stretch inadvertently. By default, the smallest of the instance height/width must be 100 screen pixels or larger to show and activate stretch handles. This variable can be set to provide a different threshold.

This variable tracks the value of the **Instance min. pixel size for stretch handles** entry in the **PCell Control** panel from the **Edit Menu**.

PCellKeepSubMasters
Value: boolean.
When a parameterized cell (pcell) is instantiated, a sub-master cell is created in memory which represents the instantiation for its given parameter set. By default, sub-master cells exist only in memory, and are created as needed from the original pcell.

When this variable is set, sub-masters that have been created will be included when writing output. This will also be true when the **StripForExport** variable is set. This applies when writing all output, **except** when using the **Save** and **Save As** buttons in the **File Menu**, and the equivalent text accelerators and including the prompts when exiting the program. It is also ignored when using the `Save` script function.

When opening a layout containing pcell instances and the corresponding cell files are found, the cell files will be read instead of evaluating the pcell. This can be faster, and it also allows the design to be opened if the original pcell is not available or can't be processed. However, the cells will behave like normal cells, not pcells, in this case.

This variable tracks the state of the **Include parameterized cell sub-masters** check box in the **Export Control** panel.

PCellListSubMasters
Value: boolean.
When a parameterized cell (pcell) is instantiated, a sub-master cell is created in memory which represents the instantiation for its given parameter set. By default, sub-master cells exist only in memory, and are created as needed from the original pcell.

When this variable is set, sub-masters that have been created will be included in the list of modified cells contained in the **Modified Cells** pop-up, which is obtained from the **Save** button in the **File Menu**. The sub-masters can be saved as native cell files in the current directory.

When opening a layout containing pcell instances and the corresponding cell files are found, the cell files will be read instead of evaluating the pcell. This can be faster, and it also allows the design to be opened if the original pcell is not available or can't be processed. However, the cells will behave like normal cells, not pcells, in this case.

This variable tracks the state of the **List sub-masters as modified cells** check box in the **PCell Control** panel.

PCellScriptPath
Value: string.
This variable provides a search path (see 2.6) to use when locating parameterized cell (pcell) scripts.

This applies when a pcell pc_script property uses the @READ directive to obtain the corresponding script, and the path provided by the directive is not rooted.

Unlike the main search path variables described in E.3, this variable is unset by default.

PCellShowAllWarnings
Value: boolean.
During pcell script evaluation, certain warning messages are disabled, including checking for coincident objects. Some of the Ciranova example pcells produce such warnings, and it is highly annoying that the messages pop up after every evaluation. The warnings may be of interest to the pcell author, but are generally nothing but a nuisance to the pcell user. If this variable is set, then these warnings will be displayed and not suppressed.

This variable tracks the state of the **Show all evaluation warnings** check box in the **PCell Control** panel.

E.8 Standard Vias

These variables apply to standard vias (see 5.8).

ViaKeepSubMasters
Value: boolean.
When a standard via is instantiated, a sub-master cell is created in memory which represents the instantiation for its given parameter set. By default, sub-master cells exist only in memory, and are created as needed by *Xic*.

When this variable is set, sub-masters that have been created will be included when writing output. This will also be true when the **StripForExport** variable is set. This applies when writing all output, **except** when using the **Save** and **Save As** buttons in the **File Menu**, and the equivalent text accelerators and including the prompts when exiting the program. It is also ignored when using the Save script function.

This variable tracks the state of the **Include standard via cell sub-masters** check box in the **Export Control** panel.

ViaListSubMasters
Value: boolean.
When a standard via is instantiated, a sub-master cell is created in memory which represents the instantiation for its given parameter set. By default, sub-master cells exist only in memory, and are created as needed by *Xic*.

When this variable is set, sub-masters that have been created will be included in the list of modified cells contained in the **Modified Cells** pop-up, which is obtained from the **Save** button in the **File Menu**. The sub-masters can be saved as native cell files in the current directory.

E.9 Scripts

The following **!set** variables affect the script parser.

LogIsLog10
Value: boolean.

In *Xic* releases prior to 3.2.23, the `log` function returned the base-10 logarithm. This definition was changed in 3.2.23, and the `log10` function added, for consistency with programming languages, *WRspice*, and most other software. This will require users to update legacy scripts that use the `log` function to call `log10` instead.

This variable provides a temporary fix. When set, the `log` function will return the base-10 value. However, it is strongly recommended that legacy scripts be updated, and this variable not be used permanently.

See also the ScriptPath variable in E.3.

E.10 Selections

The following **!set** variables affect object/cell selections using the pointing device.

MarkInstanceOrigin

Value: boolean.
When set, selected physical instances will have the cell origin marked with a cross. This applies to the selection highlighting, as well as to the ghost rendition which is attached to the mouse pointer during a move or copy operation.

Showing the origin may seem trivial, but marking the origin requires a bit of overhead since it requires running a transformation and keeping track of an additional redisplay area since the origin may be outside of the cell bounding box. Thus, the default is to not show the mark.

This variable tracks the state of the **Show origin of selected physical instances** check box in the **Selections** page of the **Window Attributes** panel from the **Set Attributes** button in the **Attributes Menu**.

MarkObjectCentroid

Value: boolean.
In mathematics, the centroid or geometric center of a two-dimensional region is the arithmetic mean of all the points in the shape. When this check box is set, selected objects will mark the centroid with a cross. This applies to the selection highlighting, as well as to the ghost rendition which is attached to the mouse pointer during a move or copy operation.

This variable tracks the state of the **Show centroids of selected physical objects** check box in the **Selections** page of the **Window Attributes** panel from the **Set Attributes** button in the **Attributes Menu**.

SelectTime

Value: integer 100–1000.
When button 1 is used for object manipulation and editing, there is a time delay which differentiates a “click” from a “drag”. This delay, which defaults to 250 milliseconds, can be adjusted by setting this variable. If the user encounters difficulty establishing an area select, for example, as opposed to a move/copy operation, then setting a longer time delay may be advantageous.

NoAltSelection

Value: boolean.
When set, the legacy logic is used for mouse click selection operations.

MaxBlinkingObjects

Value: integer 500–250000.

This can be set to an unsigned integer in the range 500–250000. If there are more than this number of objects selected, they won't blink in true-color display modes. If NoPixmapStore is set, this threshold is divided by 8. The default if not set is 25000 objects. If there are too many objects, the time to redraw for blinking becomes excessive, this variable can be used to fine-tune this threshold to the user's graphical system.

E.11 Side Menu Commands

The following **!set** variables affect the functioning of commands found in the side menu.

MasterMenuLength

Value: integer 1–75.
This integer variable sets the length of the list of master cells retained in the **Cell Placement Control** panel. The default is 25, which may not be fully visible for some screen resolutions.

This tracks the setting of the **Maximum menu length** entry in the **Cell Placement Control** panel from the side menu.

DevMenuStyle

Value: integer 0–2.
This variable tracks and sets the presentation style of the device menu (described in 7.5) which is used in electrical mode. There are three styles, selected by giving this property a value of 0, 1, or 2. The default menu, style 0, contains a menu bar with entries for categories, such as **Sources** and **Terminals**. Style 1 is similar, however the entries are alphabetic corresponding to the first letter of the device name. Style 2 provides buttons marked with the device schematic symbol. This style occupies the most screen space, but may be more convenient for new users.

LabelDefHeight

Value: real 0.01 – 10.0.
This sets the minimum label height, in microns, for new text labels. The actual initial height may be larger, depending on the zoom factor of the window, but it can not be smaller. The default if this variable is not set is 1.0 micron.

This variable was named DefLabelHeight in releases prior to 4.2.14.

This variable tracks the **Default minimum label height** entry area in the **Labels** page of the **Window Attributes** panel from the **Set Attributes** button in the **Attributes Menu**.

LabelMaxLen

Value: integer >= 6.
This variable sets the maximum width, in default-sized character cells, of a displayed label. If the label exceeds this width, it is not shown, and a small box at the text origin is shown instead. The default is 256, so this is unlikely to be triggered unless the user resets the value.

The "hidden" status of a label can be toggled by clicking the text or box with button 1 with the **Shift** key held. See 7.9 for more information.

This variable was named MaxLabelLen in releases prior to 4.2.14.

This variable tracks the **Maximum displayed label length** entry area in the **Labels** page of the **Window Attributes** panel from the **Set Attributes** button in the **Attributes Menu**.

LabelMaxLines

Value: integer >= 0.
Label text strings may have embedded newline characters which cause them to be displayed on

multiple lines. This variable, when set to a positive integer value, provides a limit on the number of lines that are actually displayed, in labels that respect this limit. Only the first N lines would actually appear in the display, where N is given in this property. If N is zero, there is no limit.

Labels observe this limit only if an internal flag is set in the label. Presently, this is set internally for the labels associated with value and param properties. The user can apply the limit to any label by setting the LIML flag in the XprpXform pseudo-property.

This variable was named MaxLabelLines in releases prior to 4.2.14.

This variable tracks the state of the **Label optional displayed line limit** numerical entry in the **Labels** page of the **Window Attributes** panel from the **Set Attributes** button in the **Attributes Menu**.

LabelHiddenMode

Value: integer 0–3.
By default, all labels participate in a protocol whereby clicking on the label with the **Shift** key held will "hide" the label, displaying a small box instead. **Shift**-clicking on the box will return to the display of the label text. This variable limits the labels which will participate in this protocol.

0	All labels, the default, same as if not set.
1	Only electrical-mode labels.
2	Only electrical-mode bound property labels.
3	No labels.

This variable was named HiddenLabelMode in releases prior to 4.2.14.

This variable tracks the state of the **Hidden label scope** option menu in the **Labels** page of the **Window Attributes** panel from the **Set Attributes** button in the **Attributes Menu**.

In the 3.2 branch of *Xic* and earlier, the default was effectively 2.

LogoEndStyle

Value: integer 0–2.
This sets the path end style used to render vector text in the **logo** command. The variable should be set to 0 for flush ends, 1 for rounded ends or 2 for extended ends. If unset, extended ends are used. This variable tracks the setting in the **Logo Font Setup** panel in the **logo** command.

LogoPathWidth

Value: integer 1–5.
This sets the relative path width used for rendering with the vector font in the **logo** command. The variable should be set to an integer 1–5, where 1 represents the smallest width, and increasing values makes the rendering appear increasingly bold. This variable tracks the setting in the **Logo Font Setup** panel in the **logo** command. If not set, a value of 3 is assumed.

LogoAltFont

Value: integer 0–1.
When set to 0 (zero), the **logo** command will use an internal bitmap font, and characters will be rendered as Manhattan polygons. When set to 1, the **logo** command will use the system font named in the LogoPrettyFont variable, or a default if this is not set. Characters are rendered as Manhattan polygons derived from the font bitmaps. When unset, or the value is not recognized, the **logo** command will use the vector font, for rendering using wires. The status of this variable tracks the check boxes in the **Logo Font Setup** panel of the **logo** command.

LogoPrettyFont

Value: font name string.

This variable sets the name of the "pretty" font to be used for text rendering in the **logo** command. It is set by the font selection panel produced from the **Select Pretty Font** button in the **Logo Font Setup** panel in the **logo** command.

Under Unix/Linux, in GTK1 releases this variable can be set to the X font description name of an X font. In GTK2 releases, a Pango font description string is expected. Under Windows, the variable is set to a string in the form "*face_name pixel_height*" or the deprecated form "(*pixel_height*)*face_name*". Examples are "`Lucida Console 24`" or "`(24)Lucida Console`", which is the default font.

LogoPixelSize

Value: positive real number <= 100.0.
When this variable is set to a value, it represents the size in microns of a "pixel" used in the **logo** command for new labels and images. With the variable defined, the "pixel" size is fixed, and can not be changed with the arrow keys from the **logo** command. This variable is set from and tracks the **Define "pixel" size** check box and text entry area in the **Logo Font Setup** panel.

LogoToFile

Value: boolean.
If this variable is set, physical text created with the **logo** command will be placed in a cell, which is instantiated at the label locations. A native cell file containing the cell is written in the current directory. If unset, the physical text is placed directly in the current cell. The variable tracks the state of the check box in the **Logo Font Setup** panel.

NoConstrainRound

Value: boolean.
When this boolean is set, there is no checking for minimum feature size of round objects as these objects are being created (they will still be tested when completed if interactive DRC is enabled).

RoundFlashSides

Value: integer 8–256, default 32.
This variable sets the number of sides per 360 degrees to use in round objects in physical mode, as created with the **round**, **donut** and **arc** side menu buttons, and corresponding script functions. It can be set from the **sides** button in the physical side menu.

ElecRoundFlashSides

Value: integer 8–256, default 32.
This variable sets the number of sides per 360 degrees to use in round objects in electrical mode, as created with the **arc** button in the menu produced by the **shapes** button in the electrical side menu. It can be set from the **sides** button in the same menu.

SpotSize

Value: real 0–1.0.
When an e-beam mask is written, the layout is rendered using a certain pixel size. This implies a mask resolution, usually cited as the "manufacturing grid" or "spot size". This size may range from 0.5 microns for the least expensive masks, down to a few nanometers for the most expensive.

Xic has two parameters which deal directly with mask resolution. The `MfgGrid` set in the technology file will force the grid snap points to be multiples of the value given. The SpotSize variable controls use of a numerical preconditioner for tiny round objects. The preconditioning should cause the pixel area to be constant with respect to positioning and rotation. This is valuable to researchers fabricating circular Josephson junctions using inexpensive mask sets (for example).

In "rasterizing" round objects to the e-beam grid, there can be numerical problems. Since the round object is rendered as a collection of spot-pixels, the feature is not particularly round, but most importantly the number of pixels used may not be well defined, and therefor the figure area may not be as expected, or consistent.

The internal spot size is used when creating round (disk) objects and donuts, but not arc objects or general polygons. It applies to the **round** and **donut** buttons in the side menu, and the corresponding script functions, but does not apply to the **arc** button or general polygons. The internal spot size is also used as the default value for the **!tospot** command.

If the SpotSize variable is given a non-negative value, this value is used as the internal spot size. The value is in microns, and 1.0 micron is the largest accepted value. If this is zero, then no preconditioning is applied. If the SpotSize variable is unset, the internal spot size will default to the `MfgGrid` given in the technology file. Thus, when a manufacturing grid is given, the default is to use preconditioning when creating round objects. This can be suppressed by setting SpotSize to zero. Other than this, there probably is no reason to set the SpotSize variable, since it should match the `MfgGrid`¿, unless the user has special requirements.

When the internal spot size has a positive value, objects created with the **round** and **donut** buttons will be created so that all vertices are placed at the center of a spot (i.e., in the center of a manufacturing grid cell), and a minimum number of vertices will be used. The **sides** number is ignored. This applies only to figures with minimum radius 50 spots or smaller; the regular algorithm is used otherwise. An object with this preconditioning applied should translate exactly to the e-beam grid. The figures are symmetric with regard to rotations in multiples of 45 degrees.

E.12 SPICE Interface

The following **!set** variables affect the interface to the *WRspice* simulator, and SPICE output in general.

SpiceListAll
: **Value:** boolean.
When set, all devices and subcircuits in the schematic will be included in SPICE output. Otherwise, only devices and subcircuits that are "connected" will be included, as explained in the **deck** and **run** command descriptions.

SpiceAlias
: **Value:** string.
This variable is set to a string which will modify the printing of device names in SPICE output. The aliasing operates on the first token of device lines. The format of the value string is

prefix1=newprefix1 prefix2=newprefix2 ...

This will cause lines beginning with *prefix* to have *prefix* replaced with *newprefix*. If the "=*newprefix*" is omitted, that line will not be printed. For example, to map all devices that begin with 'B' to 'J', and to suppress all 'G' devices, the full command is

`!set SpiceAlias B=J G`.

Note that there can be no space around the '='. After this command is given, the indicated mappings will be performed as SPICE text is produced.

SpiceHost
: **Value:** host name string.
This will set the name of the host which maintains a server for remote *WRspice* runs. If set, this will override the value of the SPICE_HOST environment variable. The host name specified in the

SPICE_HOST environment variable and the SpiceHost **!set** variable can have a suffix "`:`*portnum*", i.e., a colon followed by a port number. The port number is the port used by the `wrspiced` program on the specified server, which defaults to 6114, the IANA registered port for this service. If the server uses a non-standard port, and the `wrspice/tcp` service has not been registered (usually in the `/etc/services` file) on this port, the port number must be provided.

SpiceHostDisplay

Value: X display string.

This variable can be set to the X display string to use on a remote host for running *WRspice* through a `wrspiced` daemon, from *Xic* in electrical mode. It is intended to facilitate use of `ssh` X forwarding to take care of setting up permission for the remote host to draw on the local display.

The variable is set automatically from the **!ssh** command, or can be set by hand.

When using a remote host, this specifies the X display string to use, which is needed for running graphics. If not set, a display name will be created as follows: If the local DISPLAY variable is something like "`:0.0`", the remote display name will be "*localhostname*`:0.0`". If the local DISPLAY variable is already in the form "*localhostname*`:0.0`", this is passed verbatim.

One can use `ssh` transport for the X connection on the remote system as follows. Use "`ssh -X`" to open a shell on the remote machine. Type "`echo $DISPLAY`" into this window, it will print something like "`localhost:10.0`". Use this value for SpiceHostDisplay. The **!ssh** command will set the variable automatically. The shell must remain open while running *WRspice*, exiting the shell will close the X connection.

This will work under Windows, if Cygwin is installed, along with the OpenSSH package (for the `ssh` command) and the Cygwin X server. One weirdness: use "`ssh -Y`" instead of "`ssh -X`". The `-Y` option, which applies to recent `ssh` versions, is similar to `-X`, but overcomes stronger security checks included in recent `ssh` implementations. This seems to be necessary when using the Cygwin X server.

Background

In legacy X-window systems, the display name would typically be in the form *hostname*`:0.0`, where the *hostname* could be (and usually is) missing. A remote system will draw to the local display if the local hostname was used in the display name, and the local X server permissions were set (with `xauth`/`xhost`) to allow access. Typically, the user would log in to a remote system with `telnet` or `ssh`, set the DISPLAY variable, perhaps give "`xhost +`" on the local machine, then run X programs.

This method has been largely superseded by use of "X forwarding" in `ssh`. This is often automatic, or may require the '`-X`' or '`-Y`'option in the `ssh` command line. In this case, after using `ssh` to log in to the remote machine, the DISPLAY variable is automatically set to display on the local machine. X applications "just work", with no need to fool with the DISPLAY variable, or permissions.

The present *Xic* remote access code does not know about the `ssh` protocol, so we have to fake it in some cases. In most cases the older method will still work.

The `ssh` protocol works by setting up a dummy display, with a name something like "`localhost:10.0`", which in actuality connects back to the local display. Depending on how many `ssh` connections are currently in force, the "`10`" could be "`11`", "`12`", etc.

In the present case, if we want to use `ssh` for X transmission, the display name must match an existing `ssh` display name on the remote system that maps back to the local display.

If there is an existing `ssh` connection to the remote machine, the associated DISPLAY can be used. If there is no existing `ssh` connection, one can be established, and that used. E.g., from the `ssh` window, type "`echo $DISPLAY`" and use the value printed.

The display name provided by the SpiceHostDisplay variable will override the assumed display name created internally with the local host name.

SpiceInclude
Value: file path string.
This can be set to a file path. When a SPICE netlist is created with this variable set, text will be added to the top of the SPICE deck. If the file exists and is readable, the text from the file is added to the deck verbatim. Othewise, "`.include` *path*" is added, the *path* being the file path from the variable. This applies when creating SPICE with the **deck** button, or when preparing input for the simulator when using the **run** button.

SpiceProg
Value: program path string.
This will set the full path name of the *WRspice* executable. This is useful if there are multiple versions of *WRspice* available, or the binary has been renamed. If given, the value supersedes the values from environment variables or the **!set** variables described below.

SpiceExecDir
Value: directory path string.
This will set the directory to search for the *WRspice* executable. If given, the value overrides the SPICE_EXEC_DIR environment variable. The default search location is "`/usr/local/xictools/bin`", or, if the XT_PREFIX environment variable has been set, its value will replace "`/usr/local`".

SpiceExecName
Value: program name string.
This will set the name of the *WRspice* binary. If given, the value overrides the SPICE_EXEC_NAME environment variable. The default name is "`wrspice`".

SpiceSubcCatchar
Value: string, single printing character.
This sets the concatenation character used in *WRspice* subcircuit expansion. It affects the internally-generated node and other names within subcircuits. Please refer to the WRspice-3.2.15 release notes or documentation for a full description of the *WRspice* subc_catmode and subc_catchar variables and their effects.

SpiceSubcCatmode
Value: string, "`wrspice`" or "`spice3`".
This sets the algorithm used by *WRspice* for subcircuit expansion. It affects the internally-generated node and other names within subcircuits. Please refer to the WRspice-3.2.15 release notes or documentation for a full description of the *WRspice* subc_catmode and subc_catchar variables and their effects.

When running *WRspice* from *Xic*, there should not be compatibility issues, as *Xic* will automatically recognize the capabilities of the connected *WRspice* and compensate accordingly – as long as the hypertext facility is used to define node, branch, and device names. This is true when point-and-click is used to generate names. However, subcircuit reference names that for some reason are entered by hand may need to be updated, or a `.options` line added as a spicetext label, or the SpiceSubcCatchar, SpiceSubcCatmode variables may be set to enforce backward compatibility.

CheckSolitary
Value: boolean.
If set, warning messages will be issued when electrical netlists are generated for nodes having only one connection. This affects the **run** and **deck** commands, and the **Dump Elec Netlist** command in the **Extract Menu**.

NoSpiceTools

Value: boolean.

When running *WRspice* from *Xic*, by default the *WRspice* toolbar is shown, if *WRspice* is running on the local machine. This gives the user much greater flexibility and control over *WRspice*. If this variable is set, *before* the connection to *WRspice* is established, the toolbar will not be visible.

In releases 3.0.8 and later, this variable will also control toolbar visibility if the `wrspiced` daemon is used. However, this requires `wrspiced` distributed with wrspice-3.0.7 or later. **If this variable is set with an earlier `wrspiced` release, the *WRspice* connection will not work!**

E.13 File Menu — Printing

The following **!set** variables affect the commands in the **File Menu**, mostly the **Print** command.

NoAskFileAction

Value: boolean.

By default, in the **File Selection** and **Path Files Listing** windows, a confirmation pop-up will appear before move/copy/link operations on files or directories initiated by drag/drop. If this variable is set, this confirmation will not appear. The confirmation default is safer, but may be annoying to experienced users.

Note: in releases prior to 3.0.0, there was no confirmation, as if this variable were set.

DefaultPrintCommand

Value: string.

Under Unix/Linux/OS X, this variable overrides the default operating system command string to print a file. In Windows, this will be the printer name instead.

This should probably be set before the **Print** panel is used for the first time, as some drivers may copy the initial contents so that changing this variable will have no effect. It can be set in a startup file.

If not set, the default print command is "`lpr`" (or "`default`" in Windows). See the man page for `lpr` or `lp` for the print options which apply on your system, which can be placed in the default string. In the printer command string, the characters "`%s`" are replaced with the name of the temporary file to be printed. If these characters don't appear, the file name is tacked on the end of the command string, separated by space.

NoDriverLabels

Value: boolean.

The PostScript hard copy drivers use PostScript text for labels by default, not the vector font used on-screen. This can be overridden, and the vector font used, by setting this variable. Multi-line labels are always drawn with the vector font, however.

RmTempFileMinutes

Value: integer 0–4320.

When a layout or page is printed, a temporary file is produced and saved in one of the system temporary directories. By default, these files are not removed. The temporary directories are generally cleared when the system is rebooted, or by some other system-level means.

On some operating systems, the print command can include an option to delete the temporary source file after the print job is complete. The DefaultPrintCmd variable can be set to include this option.

Otherwise, this variable can be set to delete the temporary file a number of minutes after the print job is submitted. On some systems, the temporary file is copied into the print job queue, so that the temporary file can be deleted almost immediately. On other systems, or for large files, a link into the queue is created instead, so that the file must not be deleted until the job is complete. There is no universal way to determine if a print job has completed, so we need to wait a reasonable length of time before deleting the file.

This variable can be set to the number of minutes to wait before deleting the temporary file. If set to 0, the file will not be deleted by this system, as is the case if this variable is not set. The deletion will occur whether or not the application is still running.

Currently, this feature is not available on Windows. It uses the Unix `at` command (see the manual page for details). The user must have permission established for this to work. A message is printed in the console when a file is scheduled for deletion, or if an error (such as lack of permission) occurs.

E.14 Cell Menu Commands

The following **!set** variables affect commands found in the **Cell Menu**.

ContextDarkPcnt

Value: integer 1–100.
While the **Push** command is active, and the surrounding context is being shown, the context is drawn with reduced illumination intensity so that objects in the current cell can be visually differentiated. The variable allows the context intensity to be adjusted, as a percentage of the "normal" intensity.

If this variable is not set, a value of 65 (percent) will be used.

This variable tracks the state of the **Push context display illumination percent** entry field in the **Window Attributes** panel from the **Set Attributes** button in the **Attributes Menu**.

E.15 Editing General

The following **!set** variables affect general operations and parameters that apply during editing.

AskSaveNative

Value: boolean.
When set, the user will be prompted to save the current cell if the cell is modified, and would be saved as a native symbol, and a new current cell is about to be set. This was standard behavior in releases earlier than generation 4. Although it is always a good idea to save work periodically, the prompt can be annoying to experienced users and is now disabled by default. The user will be given the chance to save modified cells when exiting *Xic* in any case.

The **Prompt to save modified native cells** check box in the **Editing Setup** panel from the **Edit Menu** tracks the state (set or unset) of this variable.

Constrain45

Value: boolean.
When this boolean variable is set, wire and polygon vertices are constrained to form angles of multiples of 45 degrees. By default, a "smart" path generator is employed, which will construct a valid path to the pointer location from the previous point during wire or polygon construction.

This will often add two vertices: a 45 degree extension, followed by a Manhattan extension, in order to connect the points. If the **Ctrl** key is held while the new point is defined, the "smart" feature is disabled, and only one new vertex is added. If the **Shift** key is held, then the 45 degree constraint is removed entirely.

When set, rotation angles available in the **spin** command, and translation angles in the **Stretch** command, and the vertex editors for polygons and wires, are constrained to multiples of 45 degrees. However, pressing the **Shift** key will remove the constraint in these commands while the key is held. If the Constrain 45 variable is not set, holding **Shift** will impose the 45 degree angle constraint. Thus, the **Shift** key inverts the effective state of the Constrain 45 variable in these commands.

The **Constrain angles to 45 degree multiples** check box in the **Editing Setup** panel from the **Edit Menu** tracks the state (set or unset) of this variable.

NoMergeObjects

Value: boolean.
This variable tracks the state of the **Merge new boxes and polys with existing boxes/polys** check box in the **Editing Setup** panel from in the **Edit Menu** in a logically inverted sense.

By default, when a new box or polygon object is created in the database from the commands in the side menu, the new object is merged with existing boxes and polygons on the same layer, if any touch or overlap, to form a (generally more complex) polygon in the database. New wires will link with existing similar wires in the database that share an endpoint.

If this boolean variable is set, this merging will be disabled. Merging can also be disabled on a per-layer basis with the `NoMerge` technology file keyword, which prevents merging in all cases on a layer.

The NoMergePolys variable can be set to prevent merging of polygons, and will thus revert the merging behavior to that of releases prior to 3.1.7.

When reading data from a layout file, a different box clipping/merging capability is controlled by the **Clip and merge overlapping boxes** setting in the **Setup** page or the **Import Control** panel, and the corresponding MergeInput variable.

NoMergePolys

Value: boolean.
When auto-merging new objects (NoMergeObjects is not set), only boxes will be clipped and merged, polygons will be ignored, if this variable is set. This reverts to the behavior of releases prior to 3.1.7.

This variable tracks the state of the **Clip and merge new boxes only, not polys** check box in the **Editing Setup** panel from the **Edit Menu**.

NoFixRot45

Value: boolean.
There are two modes when rotating boxes/polys by non-Manhattan angles. The default and legacy method is to use an offset technique referenced to the lower-left box coordinate, or the first vertex of polygons. This ensures that the same figure is generated at any location, and seems to ensure that all angles are exactly multiples of 45 or 90, after rotation, in boxes. However, this has the problem that two figures that abut before rotation might no longer abut after rotation. For example, use the **!split** function to split a disk, then rotate the collection by 45 degrees. It is likely that some of the figures no longer touch. If merging is enabled, the disk will have lines through it at these points, where the gaps prevented merging.

If NoFixRot45 is set, the offset fix is not done. This solves the problem of gaps appearing between rotated objects, but has its own problems. Namely, rectangles aren't preserved, angles can differ from 45s. Try rotating a small rectangle, say 3x5 internal units, by 45s in this mode, and one can see it is a mess. Larger rectangles are not visually distorted, but there are 1-unit errors in the vertex placements relative to preservation of 45s or 90s. This is probably not acceptable for most work.

Really, rotating by 45 degrees is something best avoided.

E.16 Edit/Modify Menu Commands

The following **!set** variables affect commands found in the **Edit Menu** and the **Modify Menu**.

UndoListLength
: **Value:** integer >= 0.
This variable sets the number of operations remembered in the **Undo** command. If not set, 25 operations are saved. If set to zero, the length is unlimited.

This tracks the setting of the **Maximum undo list length** entry area in the **Editing Setup** panel from the **Edit Menu**.

MaxGhostDepth
: **Value:** integer 0–8.
This variable sets the maximum expansion depth for instance expansion in ghosting. If not set, this is the same as the normal expansion depth. The actual expansion depth used in ghosting will not be larger than the normal expansion depth, but can be smaller. For example, setting this to 0 (zero) will prevent expansion of ghosted subcells entirely.

This tracks the setting of the **Maximum subcell depth in ghosting** menu in the **Editing Setup** panel from the **Edit Menu**.

MaxGhostObjects
: **Value:** integer 50–50000.
This sets the maximum number of objects to render individually as "ghosts" attached to the mouse pointer during operations such as move and copy. This can be set to an unsigned integer in the range 50–50000. If there are more than this number, some outlines won't be shown, the smaller-area objects will be skipped. If subcells are being expanded, objects are rendered top-down, so that if the limit is reached, objects deeper in the hierarchy will not be shown.

The default is 4000 if this variable is not set. If, when moving a large number of objects, the pointer motion is too sluggish, the user can set this variable to compensate, or can limit the subcell expansion depth by setting MaxGhowtDepth if expansion causes the problem.

This tracks the setting of the **Maximum number of ghost-drawn objects** entry area in the **Editing Setup** panel from the **Edit Menu**.

NoWireWidthMag
: **Value:** boolean.
When set, the width of wires does not change when the wire undergoes magnification, in a **Move**, **Copy**, or **Flatten** operation.

The **No wire width change in magnification** check box in the **Editing Setup** panel from the **Edit Menu** tracks the state (set or unset) of this variable.

CrCellOverwrite
: **Value:** boolean.
When set, The **Create Cell** operation in the **Edit Menu** and the `CreateCell` script function can overwrite cells already in memory. This can be dangerous and is prevented by default.

The **Allow Create Cell to overwrite existing cell** check box in the **Editing Setup** panel from the **Edit Menu** tracks the state (set or unset) of this variable.

LayerChangeMode
Value: tri-state.
This variable applies during all move and copy operations, and during the **spin** command in the physical side menu and similar. In these commands, when objects being moved or copied are ghost drawn as attached to the mouse pointer, it is possible to change the current layer. The operation is then completed by clicking at the new location in a drawing window.

This is a tri-state variable. If not set, there will be no layer change in these commands. Thus by default any current layer change made during the command is ignored by the command. If set to the string "`all`" (case insensitive), then a layer change will apply to all objects being moved or copied. All new objects will be placed on the new layer, regardless of the original layers of the objects. If set to anything else, including to nothing (i.e., as a boolean) then only objects on the previous current layer will be changed to the new layer. Other objects will remain on their original layer.

This variable tracks the state of the radio buttons in the **Layer Change Mode** pop-up, which appears when the **Set Layer Chg Mode** button in the **Modify Menu** is pressed.

JoinMaxPolyVerts
Value: integer 0 or 20–8000.
This variable applies to the **Join** and **Join All** buttons in the **Join or Split Objects** panel (from the **Edit Menu**), the **!join** command, the join (merging) operation when new objects are created, and the associated script functions and elsewhere where join operations occur.

This sets an upper bound on the number of vertices in polygons created by a join operation. The default is 600 vertices. If set to 0, no limit is applied. The variable tracks the **Maximum vertices in joined polygon** entry in the **Join or Split Objects** panel.

There is no internal limit on the vertex count of a polygon in memory. Although setting JoinMaxPolyVerts to 0 allows arbitrarily large polygons to be created, one should be reasonable. Huge polygons can be cumbersome and inefficient. Oversize polygons and wires will be broken up, if necessary, when a file is saved to disk. For the different formats, the limits are

native	no limit
CIF	no limit
CGX	8000 vertices
GDSII	depends on GdsOutLevel, max is 8000 vertices
OASIS	no limit

For CIF files, *Xic* can read/write arbitrarily large polygons and wires, but beware that other tools may have built-in limits.

JoinMaxPolyGroup
Value: integer >= 0.
This variable applies to the **Join** and **Join All** buttons in the **Join or Split Objects** panel (from the **Edit Menu**), the **!join** command, the join (merging) operation when new objects are created, and the associated script functions and elsewhere where join operations occur.

When a collection of trapezoids is being combined into polygons during a join operation, the collection is first divided into connected groups, each of which will be converted to one or more polygons. This variable limits the number of trapezoids in the groups. The default value (when this variable is unset) is 0, meaning that there is no limit. Generally, applying a limit (for example, 300) provides faster join operations, however this will leave as separate objects more polygons that could have been joined.

This variable tracks the **Maximum trapezoids per poly for join** entry in the **Join or Split Objects** panel.

JoinMaxPolyQueue
Value: integer >= 0.
This variable applies to the **Join** and **Join All** buttons in the **Join or Split Objects** panel (from the **Edit Menu**), the **!join** command, the join (merging) operation when new objects are created, and the associated script functions and elsewhere where join operations occur.

When objects are being joined, they are first decomposed into trapezoids. The trapezoids from the objects are saved in a single list, and when the list length exceeds a certain value the list is sent to the function that recombines the trapezoids into polygons. This variable is used to set the length threshold. The default value (when this variable is unset) is 0, which allows the list to grow without bound. Generally, applying a limit (for example, 1000) provides faster processing, but will produce more polygons.

This variable tracks the **Trapezoid queue size for join** entry in the **Join or Split Objects** panel.

JoinBreakClean
Value: boolean.
This variable applies to the **Join** and **Join All** buttons in the **Join or Split Objects** panel (from the **Edit Menu**), the **!join** command, the join (merging) operation when new objects are created, and the associated script functions and elsewhere where join operations occur.

In a join operation, when building up the polygons and the vertex limit (JoinMaxPolyVerts) is reached, ordinarily the present polygon is output, and a new one is started immediately. This generally produces a set of polygons with complicated and seemingly arbitrary borders. If this variable is set, then the polygons are initially built ignoring the vertex limit, and polygons that exceed the vertex limit are split into pieces along Manhattan bisectors, so that no piece exceeds the vertex count. This gives a much nicer looking layout, but is more compute intensive.

This variable tracks the **Clean break in join operation limiting** check box in the **Join or Split Objects** panel.

JoinSplitWires
Value: boolean.
This applies to join operations as listed for the variables above, but not for the joining when new objects are created. It also applies to the split operation.

By default, wires do not participate in join/split operations, these operate on boxes and polygons only. Wires, however, will be joined with other wires on the same layer it they share an endpoint and have the same width. If this variable is set, then wires will be treated like polygons in join and split operations, but wires never participate in the join operation when new objects are created.

This variable tracks the **Include wires (as polygons) in join/split** check box in the **Join or Split Objects** panel.

PartitionSize
Value: floating-point number.
This variable applies to layer expression evaluation, including from the **Evaluate Layer Expression** panel (from the **Edit Menu**), the **!layer** and **!compare** commands, and the `AdvanceZref` script function.

In releases prior to 3.0.0, this variable was named “LayerPartSize”.

When geometrical operations are performed over a large area, a logical square grid is created over the area relative to the lower-left corner. The operations are performed for each grid element that intersects the area, and the results are combined. This can be more efficient than performing the operations over the entire area in one shot. Performance rapidly degrades as the amount of geometry per grid area increases. Best performance is probably obtained with 10000 or fewer trapezoids per grid.

This variable specifies the size of the grid, in microns, set as a floating-point number. If not set, the default grid size is 100 microns. Acceptable values are 1.0 – 10000.0, or 0. If set to 0, partitioning is not used.

The variable tracks the **Partition size** set in the **Evaluate Layer Expression** panel.

Threads

Value: integer 0–31.
PRELIMINARY, EXPERIMENTAL!

This will enable new multi-threaded functionality as it becomes available. This is set to the number of helper threads that can be called upon to parallelize certain operations. The best value is probably one less than twice the number of available processor cores. It should not be set to a larger value, but one might wish to try smaller values. If unset, or set to 0, the program is single threaded.

This variable tracks the **Number of helper threads** entry in the **Evaluate Layer Expression** panel from the **Edit Menu**.

Presently, multi-threading is used when evaluating a layer expression using a grid. Evaluation in each of the grid cells can be done in parallel, so these jobs are submitted to the thread pool. One can experiment with the partition size to get fastest results, larger partitions are more likely to overcome the multi-threading overhead.

E.17 View Menu Commands

The following **!set** variables affect commands found in the **View Menu**.

InfoInternal

Value: boolean.
When set, the **Info** command in the **View Menu** and the **Info** command in the **Cells Listing** panel will print dimensions using internal database units (default is 1000 per micron) rather than in microns.

PeekSleepMsec

Value: integer >= 0.
This sets the delay time in milliseconds to wait after a layer is drawn in the **Peek** command. The default is 400.

LockMode

Value: boolean.
This variable, when set, locks the current mode (physical or electrical). In addition, while reading any type of file, only the information for the present mode is read. All features which apply to the other mode are disabled, and no data are stored for the other mode. By not storing stubs for the electrical data, for example, more memory space is available for a large physical-only file.

As files written from this mode have only one type of data, it is possible to overwrite files that originally contained both types of data. The user should be aware of this possibility.

XSectNoAutoY

Value: boolean.

By default, the cross-section display is shown with a vertical scale adjusted such that the entire layer stack occupies most of the window. This is maintained independent of the window magnification, which consequently changes only the X-scale. If this boolean variable is set, the auto-scaling will not be done.

This variable is set by the **Auto Y-Scale** check box that appears in the **Set Display Window** pop-up that is called by the **Zoom** button in the **View** menu of the cross-section display window. The setting is done only when the user presses the **Apply** button.

XSectYScale

Value: real 1e-3 – 1e3.

This variable supplies a Y-scale to the cross-section display. If the auto-scaling is enabled, the scale factor determines how much of the vertical window dimension is occupied by the layer stack. Without auto-scaling, this scale is applied directly to the vertical axis.

The horizontal grid lines and ruler gradations take into account the scale. The scaling allows easy visualization when the thickness is much larger or much smaller than typical line widths.

This variable is set from the **Y-Scale** numerical entry area that appears in the **Set Display Window** pop-up that is called by the **Zoom** button in the **View** menu of the cross-section display window. The setting is done only when the user presses the **Apply** button, and the value has been set to something other than unity.

E.18 Attribute Menu Commands

The following **!set** variables affect the commands found in the **Attributes Menu**.

TechNoPrintPatMap

Value: boolean.

When set, *Xic* will use the hex format when writing stipple patterns for layers when writing a technology file. If unset, an ASCII format, that provides a rendition of the map, is used. The hex format is compatible with *Xic* releases prior to 3.2.25, if the stipple map sizes are restricted to 8x8, 16x8, 8x16, or 16x16. Technology files can be written using the **Save Tech** button in the **Attributes Menu**.

TechPrintDefaults

Value: boolean or string.

When a technology file is written with the **Save Tech** button, by default entries that would set a parameter to a program default value are omitted, as they are redundant and increase the size and complexity of the file. This will be the case when this variable is not set. If this variable is set to no value, i.e., as a boolean, then these lines will be added to the technology file as comments. If this variable is set to any value, then these lines will be added as active text.

This variable tracks the radio buttons in the **Write Tech File** pop-up which appears from the **Save Tech** button in the **Attributes Menu**.

BoxLineStyle

Value: integer, default e38 (hex).

This sets the line style mask of the boxes used in electrical mode, and in physical mode for some highlighting purposes, such as the current cell boundary. The style is an integer whose binary value is replicated to form the lines used in the box. The line style editor in the **Grid Setup** panel can be used to generate line style masks.

The **Global Attributes** button in the **Tech Parameter Editor** provides a prompt-line interface for setting this variable. This is called from the **Edit Tech Params** button in the **Attributes Menu**.

EraseBehindProps

Value: boolean.

If given, the area inside the bounding box of text generated by the **Show Phys Properties** command in the **Main Window** sub-menu of the **Attributes Menu** or the sub-window **Attributes** menu is erased, to promote visibility of the text.

This tracks the state of the **Erase behind physical properties text** check box in the **Phys Props** page of the **Window Attributes** panel from the **Set Attributes** button in the **Attributes Menu**.

PhysPropTextSize

Value: integer 6–48.

This variable can be used to set the height, in pixels, of the text used to render physical properties on-screen when physical properties are being displayed. If not set, the default is 14.

This tracks the state of the **Physical property text size (pixels)** entry area in the **Phys Props** page of the **Window Attributes** panel from the **Set Attributes** button in the **Attributes Menu**.

EraseBehindTerms

Value: boolean or "`all`".

If set, the area inside the bounding box of terminals made visible by the **Show Terminals** command is erased, to promote visibility of the text. If set to "`all`", all terminals are erased behind, otherwise only the cell's formal terminals are erased behind.

This tracks the setting of the **Erase behind physical terminals** menu in the **Terminals** page of the **Window Attributes** panel from the **Set Attributes** button in the **Attributes Menu**.

TermTextSize

Value: integer 6–48.

This variable can be used to set the height, in pixels, of the text used in rendering terminals and cell labels in electrical mode. If not set, the default is 14.

This tracks the setting of the **Terminal text pixel size** entry in the **Terminals** page of the **Window Attributes** panel from the **Set Attributes** button in the **Attributes Menu**.

TermMarkSize

Value: integer 6–48.

This variable can be used to reset the pixel size of the cross used as a terminal mark. If not set, the default is 10.

This tracks the setting of the **Terminal mark size** entry in the **Terminals** page of the **Window Attributes** panel from the **Set Attributes** button in the **Attributes Menu**.

ShowDots

Value: boolean or "`n`" of "`a`".

This variable sets the mode used to display connection indications (dots) in schematics in electrical mode. It tracks and sets the state of the buttons in the **Connection Points** panel available from the **Connection Dots** button in the **Attributes Menu**.

If not set or set as a boolean, the normal indication is used, whereby only "ambiguous" connection points are marked. These are wire vertices common to two or more wires (except for common end vertices of two wires), non-endpoint wire vertices common with device or subcircuit terminals, and any point common to three or more terminals or wire vertices. Note that the default for release 4.3.13 and earlier was to not show connection dots.

If set to "`none`" or any word starting with '`n`' or '`n`', no connection point indication is used. If set to "`all`" of any word starting with '`a`' or '`A`', all connections are marked with a dot. This is sometimes useful to see if a connection actually exists at a given location.

The computation of dot locations can be repeated by turning dots off, then on again. This may be needed occasionally to correct misplaced dots.

FullWinCursor

Value: boolean.
When this variable is set, the default cursor consists of horizontal and vertical lines that extend completely across the drawing window. The lines intersect at the nearest snap point in the current window.

This variable tracks the state of the **Use full-window cursor** check box in the **General** page of the **Window Attributes** panel. The **Set Attributes** button in the **Attributes Menu** produces this.

CellThreshold

Value: integer 0–100.
This sets the size threshold in pixels for physical mode subcells to be shown in the display. If not set, the value is effectively 4. Subcells that are smaller than this size in the display are either shown as a bounding box, or not shown at all, depending on the setting of the **Subthreshold Boxes** button in the **Main Window** sub-menu in the **Attributes Menu** or the sub-window **Attributes** menu. If set to 0, all detail is drawn, which can significantly increase rendering time. This applies to hard copy output as well as to on-screen rendering.

This variable tracks the **Subcell visibility threshold (pixels)** entry area in the **General** page of the **Window Attributes** panel from the **Set Attributes** button in the **Attributes Menu**.

In electrical mode, the threshold is effectively fixed at one pixel.

GridNoCoarseOnly

Value: boolean.
When this boolean variable is set, as one zooms out, when the fine grid becomes so fine that it is not shown, the coarse grid will also not be shown. Otherwise, the coarse grid (only) will be shown. This tracks the state of the check box in the **All Windows** group in the **Style** page of the **Grid Setup** panel of the main window. This can be brought up with the **Set Grid** button in the **Main Window** sub-menu of the **Attributes Menu**. This applies in physical mode only, in electrical mode the coarse grid is not shown without the fine grid.

GridThreshold

Value: integer 4–40, default 8.
This sets the number of pixels that is the minimum grid spacing, in both physical and electrical modes. If the grid were to have a smaller displayed spacing, it will not be shown. Accepted values are in the range 4 – 40, and the value is taken as 8 if this variable is not set. This tracks the value of the numerical entry area in the **All Windows** group in the **Style** page of the **Grid Setup** panel for the main window. This can be brought up with the **Set Grid** button in the **Main Window** sub-menu of the **Attributes Menu**.

E.19 Convert Menu — General

Below are general variables relating to data input/output and format translation.

ChdFailOnUnresolved

Value: boolean.

If this variable is set, when doing an operation with a Cell Hierarchy Digest (CHD) that was created from a file containing unresolved references (cells that were referenced but not defined in the file), and the cells can't be referenced through libraries, the operation will fail. If not set, processing will continue, with the non-references either being ignored (e.g., when flattening), or converted to empty cells (when reading into the database), or propagated to output (when writing output), depending on the operation.

ChdCmpThreshold

Value: integer >= 0.

When using a Cell Hierarchy Digest (CHD), by default instance lists larger than 256 bytes are stored in compressed form in memory. This reduces memory use, but there is a small speed penalty.

This variable sets the size threshold for compression. If set to a value less than 100, no compression is done. Otherwise, instance lists larger than the set size (in bytes) will be compressed. Experimentation suggests that the largest blocks dominate the decompression overhead, so that the value of this variable has little effect, except when turning off compression entirely.

MultiMapOk

Value: boolean.

When set, multiple input/output GDSII layer/datatype mapping to *Xic* layers is enabled (as was always the case in *Xic* releases prior to 2.5.67-5). This allows objects in GDSII/OASIS files to be created on more than one *Xic* layer, and objects on *Xic* layers to be instantiated more than once in GDSII/OASIS output files (each with a different layer/datatype). When not set, each object is created or written once only, using the first mapping in the internal list that applies (first matching layer or StreamOut keyword found).

NoPopUpLog

Value: boolean.

When set, the **File Browser** loaded with the log file which appears if there were errors or warnings when reading an input file or writing output will *not* appear. This applies to the **Open** command and equivalent, and the file input/output operations in the **Convert Menu**. It is not recommended to set this in general, but the browser popping up does become annoying at times, so this variable can be set when the user knows what to expect in the file.

UnknownGdsLayerBase

Value: integer 0–65535.

When translating to GDSII or OASIS from a file format that does not have layer/datatype numbers, and no mapping can be resolved, new layer/datatype combinations are created. The new layer numbers are generated sequentially, starting with the value of UnknownGdsLayerBase, or 128 if this variable is not set. Each is given the datatype UnknownGdsDatatype.

UnknownGdsDatatype

Value: integer 0–65535.

This is the datatype assigned to new layers generated using the UnknownGdsLayerBase. if not set, a datatype 128 is used.

NoStrictCellnames

Value: boolean.
If the boolean variable NoStrictCellnames is set, there will be no checking of cell names for white space, and the legacy behavior (in releases prior to 3.0.5) of accepting white space in cell names will be enabled. Otherwise, white space is not allowed in cell names, and if such cells are found in an archive being read, aliasing will be employed to map white space characters to underscores.

NoFlattenStdVias

Value: boolean.
When set, and when flattening a physical cell hierarchy, standard via instances will be retained as such rather than being converted to geometry. This variable tracks the state of the **Don't flatten standard vias, move to top** check box in the **Flatten Hierarchy** panel, and the **Don't flatten standard vias, keep as instances at top level** check boxes in the **Setup** pages of the **Import Control** panel, **Export Control** panel, and the **Format Conversion** panel.

Presently, when the input data source is an archive file, this variable applies only when sub-masters are **not** contained in the source file, and are therefor created in *Xic*.

NoFlattenPCells

Value: boolean.
When set, and when flattening a physical cell hierarchy, parameterized cell (pcell) instances will be retained as such rather than being converted to geometry. This variable tracks the state of the **Don't flatten param. cells, move to top** check box in the **Flatten Hierarchy** panel, and the **Don't flatten pcells, keep as instances at top level** check boxes in the **Setup** pages of the **Import Control** panel, **Export Control** panel, and the **Format Conversion** panel.

Presently, when the input data source is an archive file, this variable applies only when sub-masters are ¡b¿not¡/b¿ contained in the source file, and are therefor created in *Xic*.

NoFlattenLabels

Value: boolean.
When set, and when flattening a cell hierarchy (physical or electrical), labels found in subcells are ignored (not copied into the current cell). Labels found in the current cell are retained. This is intended to avoid creating conflicting net labels of wire nets from (subnet) labels in subcells. This variable tracks the state of the **Ignore labels in subcells** check box in the **Flatten Hierarchy** panel, and the **Ignore labels in subcells** check boxes in the **Setup** pages of the **Import Control** panel, **Export Control** panel, and the **Format Conversion** panel.

NoReadLabels

Value: boolean.
When this variable is set, text label elements will not be read from archive files in physical mode. This may improve efficiency if the user is concerned with physical layout data only. This variable tracks the setting of the **Skip reading text labels from physical archives** check box in the **Setup** page of the **Import Control** panel from the **Convert Menu**.

KeepBadArchive

Value: boolean.
When generating an archive file and an error occurs, the archive file will normally be deleted. However, if this variable is set, the output file will be given a "`.BAD`" extension and retained. This file should be considered corrupt, but may be useful for diagnostics.

E.20 Convert Menu — Input and ASCII Output

The **!set** variables below affect the format conversion when importing data from a file. Many of these variables have counterpart controls in the **Import Control** panel from the **Convert Menu**. The functionality also applies in many cases when input is being read in the **Open** command and similar.

The following table identifies where the variables in this section are set, if settable from the graphical interface, and specifies the scope of the variables.

Variable	**Set From**	**Notes**
ChdLoadTopOnly	**Import Control**	5
ChdRandomGzip		6
AutoRename	**Import Control**	1
NoCreateLayer	**Import Control**	1
NoAskOverwrite	**Import Control**	1
NoOverwritePhys	**Import Control**	1
NoOverwriteElec	**Import Control**	1
MergeInput	**Import Control**	1
NoPolyCheck	**Import Control**	1
DupCheckMode	**Import Control**	1
EvalOaPCells	**Import Control**	1
NoEvalNativePCells	**Import Control**	1
NoCheckEmpties	**Import Control**	1
NoReadLabels	**Import Control**	1
LayerList	layer change module	2
UseLayerList	layer change module	2
LayerAlias	layer change module	2
UseLayerAlias	layer change module	2
InToLower	cell name mapping module	3
InToUpper	cell name mapping module	3
InUseAlias	cell name mapping module	3
InCellNamePrefix	cell name mapping module	3
InCellNameSuffix	cell name mapping module	3
NoMapDatatypes	**Import Control**	1
CifLayerMode	**Import Control**	1
OasReadNoChecksum		1
OasPrintNoWrap	**Format Conversion**, **ASCII Text** page	4
OasPrintOffset	**Format Conversion**, **ASCII Text** page	4

Notes:

1. These variables apply whenever a layout file is being read, in any mode.
2. These variables apply to actions initiated from any panel containing the layer filtering/aliasing module, and to the following script functions:

```
OpenCell
FromArchive
OpenCellHierDigest
ChdEdit
ChdOpenFlat
ChdWrite
```

```
ChdWriteSplit
ChdLoadGeometry
```

3. These variables apply to actions initiated from any panel containing the **Cell Name Mapping** control group, and to the following script functions:

```
OpenCell
FromArchive
OpenCellHierDigest
```

4. These variables apply only when writing ASCII text from OASIS input.
5. These variables apply when reading cells into main memory from a Cell Hierarchy Digest.
6. These variables apply when reading gzipped GDSII or CGX files through a Cell Hierarchy Digest.

ChdLoadTopOnly

Value: boolean.

When set, when reading cells into the main database from a Cell Hierarchy Digest (CHD), only the requested cell is actually read. Any subcells of the cell become reference cells in the main database. This allows editing of the requested cell, and when written to disk the complete hierarchy will appear, however loading the whole hierarchy into memory is avoided.

This variable tracks the state of the **Load top cell only** check box in the **Cell Hierarchy Digests** panel.

ChdRandomGzip

Value: boolean or 0–255.

This variable enables use of a random-access mapping capability for Cell Hierarchy Digest (CHD) accesses to gzipped GDSII and CGX files. This will speed up CHD operations that must seek randomly in the input file.

CHDs created while this variable is set will include the mapping structure if the input file is gzipped. The mapping structure provides access points to data within the file, spaced by default by about 1Mb of uncompressed data. The map requires about 32Kb per access point. When seeking in the file, one can jump to the closest earlier access point, and read to the desired offset. Without the mapping, one can only read forward from the current location to the desired location, or rewind to the beginning and read to the desired location.

The integer is the number of Mb between access points. If 0, it is as if the variable is not set. Setting as a boolean, i.e., to no value, is equivalent to setting to 1.

AutoRename

Value: boolean.

When set, when reading archive files and a cell is encountered with the same name as a cell already in memory, the new cell name is automatically changed to avoid a clash. Thus, the **Merge Control** pop-up never appears when this variable is set. The new name has an added suffix "$\$N$" where N is an integer. When this is set, the alias file (if enabled) is never updated. A warning is added to the log file when a cell name is changed. This is part of a more general cell name mapping capability (see 14.2). This variable is set when the **Auto Rename** entry is selected in the **Default when new cells conflict** menu in the **Setup** page of the **Import Control** panel from the **Convert Menu**.

NoCreateLayer

Value: boolean.

When set, when reading an input layout file and a layer is found that can't be mapped to the existing *Xic* layers, the read will be aborted. The behavior otherwise is to create new layers as needed.

This variable tracks the state of the **Don't create new layers when reading, abort instead** check box in the **Setup** page of the **Import Control** panel from the **Convert menu**.

NoMapDatatypes

Value: boolean.

This variable affects only the creation of new layers when a GDSII or OASIS file is read. The default behavior is to create a separate new *Xic* layer for each GDSII layer/datatype encountered that is not mapped in the technology file. With the variable set, all datatypes on the new GDSII layer are mapped to the same (new) *Xic* layer. This variable tracks the state of the **Map all unmapped GDSII datatypes to same Xic layer** check box in the **Setup** page of the **Import Control** panel from the **Convert Menu**.

NoAskOverwrite

Value: boolean.

If a disk file is opened which contains a cell with the same name as one already in memory, and AutoRename is not set, the default behavior is to produce a **Merge Control** pop-up which gives the user control over how to proceed. If this variable is set, then the pop-up will not appear, and the default action will be taken. The default action can be specified with the NoOverwritePhys and NoOverwriteElec variables. This variable tracks the state of the **Don't prompt for overwrite instructions** check box in the **Setup** page of the **Import Control** panel from the **Convert menu**.

NoOverwritePhys
NoOverwriteElec

Value: boolean.

These control the default behavior when a cell from a file being read conflicts with the name of a cell already in memory. The default behavior is for the cell from the file to overwrite the cell in memory. If NoOverwritePhys is set, the physical part of the cell in memory will not be overwritten, and the physical part of the cell in the file will be ignored. Similarly, if NoOverwriteElec is set, the electrical part of the cell in memory will be preserved, and the electrical part of the cell from the file will be ignored. This variable is set according to the choice in the **Default when new cells conflict** menu in the **Setup** page of the **Import Control** panel from the **Convert Menu**.

NoOverwriteLibCells

Value: boolean.

By default, existing cells in memory can be overwritten if a cell of the same name is read when opening cells from an archive file, if the overwriting mode is enabled. Setting this variable will prevent existing cells that were opened through the library mechanism (and thus has the LIBRARY flag set) from being overwritten.

The **No Overwrite Lib Cells** button in the **Libraries Listing** pop-up tracks the state of this variable.

NoCheckEmpties

Value: boolean.

When set, there is no checking for empty cells as an input file is being read, and the pop-up that normally appears when a file is opened for editing if there are empty cells in the hierarchy is suppressed. An "empty cell" as listed is a cell that is either absent or has no content in both electrical and physical modes. It is possible to check for empty cells at any time with the **!empties** command. This variable tracks the setting of the **Skip testing for empty cells** check box in the **Setup** page of the **Import Control** panel from the **Convert Menu**.

NoPolyCheck

Value: boolean.

When this boolean variable is set, the tests for problematic conditions such as self-overlap, normally applied to polygons, is skipped. The default behavior is to check each polygon for potentially troublesome geometry specification while the polygon is being created. If a layout is known to have only "good" polygons, then turning off this test may slightly reduce reading time.

This variable tracks the setting of the **Skip testing for badly formed polygons** check box in the **Setup** page of the **Import Control** panel from the **Convert Menu**.

DupCheckMode

Value: boolean or string.

When reading layout data and identical objects or subcells are found at the same location, the default action is to issue a warning message and read the duplicates into the database. This variable can be set to alter the default behavior. If set to a word starting with '`r`' (case insensitive), the duplicate objects or subcells will not be brought into the database. As duplicates are almost always layout errors, it makes sense to filter them, though they generally cause no harm. If this variable is set to a word starting with '`w`', only a warning will be issued, exactly as if the variable were not set. If set to anything else, including an empty string (i.e., set as a boolean), testing for duplicates is disabled. This may very slightly reduce the time to read in a file.

This variable tracks the setting of the **Duplicate item handling** menu in the **Setup** page of the **Import Control** panel from the **Convert Menu**.

EvalOaPCells

Value: boolean.

When a non-native pcell placement is encountered when reading file input, the default behavior is to not attempt to evaluate the pcell, and assume that the sub-master has been exported. Generally, evaluation of a Skill-based pcell will fail, unless Virtuoso is accessible and the pcell caching has been turned on and is up to date.

If this variable is set, *Xic* will attempt to evaluate foreign pcell placements, which is necessary if the sub-masters have not been supplied by another means. The OpenAccess library that supplies the super-master must be open.

If sub-masters are available, it is faster to use them rather than to evaluate the scripts and recreate the sub-master.

This variable tracks the status of the **PCell evaluation: Eval OpenAccess** check box in the **Setup** page of the **Import Control** panel from the **Convert Menu**.

NoEvalNativePCells

Value: boolean.

When a native pcell placement is encountered when reading file input, the default behavior is to attempt to locate the super-master and evaluate the script, generating the sub-master. It is assumed therefor that the super-master is available. If the sub-masters have been included in the archive or otherwise made available, then this variable should be set. Otherwise, the super-masters must be available.

This variable tracks the status of the **PCell evaluation: Don't eval native** check box in the **Setup** page of the **Import Control** panel from the **Convert Menu**.

MergeInput

Value: boolean.

When this variable is set, and a layout file is being read into the database, boxes on the same layer are merged together, if possible, as files are being read in. Overlapping boxes are clipped and/or merged, so that in the database no boxes will overlap.

Merging will not occur on a layer with the `NoMerge` technology file keyword applied.

This variable tracks the setting of the **Clip and merge overlapping boxes** check box in the **Setup** page of the **Import Control** panel from the **Convert Menu**.

LayerList

Value: string.
This can be set to a space-separated list of layer names (see 14.4). These layers can be used for filtering when an archive file is being read or translated. Each name should be in a format which will match a layer in the file type being processed, with wildcarding allowed. This variable is part of the layer mapping and filtering capability, as used in the **Import Control** and **Format Conversion** panels, and tracks the entry area. Actual utilization of the layer list is controlled by the UseLayerList variable.

UseLayerList

Value: boolean or string.
This variable determines how and if the LayerList string is used when input is being read from an archive file. This variable is part of the layer mapping and filtering capability, as used in the **Import Control** and **Format Conversion** panels, and tracks the check boxes.

If UseLayerList in not set, the LayerList is ignored, and any layer found in the input file will be read or converted. If UseLayerList is set to a word starting with '`n`' or '`N`', layers that are listed in the LayerList will *not* be converted. If UseLayerList is set to a anything else (including no value) *only* the layers listed in the LayerList will be converted.

LayerAlias

Value: string.
This variable can be set to a string consisting of space-separated *name=value* pairs, where *name* is an existing layer name and *value* is a layer name to which *name* will be mapped during conversions, if UseLayerAlias is set.

This variable can be set from the **Layer Aliases** editor, which is available from pop-ups that control operations where layer filtering and modification is available, as in the **Import Control** and **Format Conversion** panels. The variable can also be set using script functions.

UseLayerAlias

Value: boolean.
When this variable is set, when reading an archive or native file and layer aliasing is available, layers encountered are aliased according to entries in the LayerAlias variable.

Aliasing occurs on reading only, after the LayerList is processed, if this feature is used. Thus, a LayerList used for reading should contain the unaliased layer names. Layer aliasing applies to physical data only, under conditions equivalent to those listed for UseLayerList. This variable is part of the layer mapping and filtering capability, and tracks the **Use Layer Aliases** check box, as in the **Import Control** and **Format Conversion** panels.

InToLower

Value: boolean.
When set, cell names found in archive files being read that are entirely upper case will be mapped to lower case. A name that is mixed-case will not be changed. This mapping occurs for names which are not aliased in an enabled alias file. This is part of a more general cell name mapping facility (see 14.2), available in the **Import Control** panel and elsewhere.

InToUpper

Value: boolean.
When set, cell names found in archive files being read that are entirely lower case will be mapped

to upper case. A name that is mixed-case will not be changed. This mapping occurs for names which are not aliased in an enabled alias file. This is part of a more general cell name mapping facility (see 14.2), available in the **Import Control** panel and elsewhere.

InUseAlias

Value: boolean or string.

This variable enables utilization of the alias file (see 14.3) when reading from an archive file. If simply set as a boolean, i.e., to no value, the alias file will be read before the operation, and created or updated if necessary after the operation. If the variable is set to a word starting with '`r`' (case insensitive), then the alias file will be read before the operation and used during the operation (if it exists), but will not be created or updated after the operation. If the variable is set to a word starting with '`w`' or '`s`' (case insensitive), the alias file will not be read before an operation, but will be created or updated after the operation completes. This is part of a more general cell name mapping facility (see 14.2), available in the **Import Control** panel and elsewhere.

InCellNamePrefix, InCellNameSuffix

Value: string.

These variables are most simply set to a text token that is added to the beginning or end of cell name strings as archive files are being read. Modifications will not be made to cell names found in an enabled alias file. The strings can also be given in the form

/str/sub/

where *str* and *sub* are text tokens, separated by forward slash characters as shown. In this case if the characters at the beginning/end of the cell name (for prefix/suffix) match the *str*, they are replaced by *sub*. This is the same action as is used in the **!rename** command. The string token must match exactly — there is no wildcarding. Either the prefix or suffix, or both, can be defined. The suffix substitution occurs after the prefix substitution. Either can match the whole cell name if one wants to change the name of a single cell. This is part of a more general cell name mapping facility (see 14.2), available in the **Import Control** panel and elsewhere.

CifLayerMode

Value: integer 0–2.

This variable determines how *Xic* interprets layer directives while reading CIF files. This is the same as the **How to resolve CIF layers** menu in the **Import Control** panel. Setting to 0 is the default **Try Both** option, 1 is the **By Name** option, and 2 is the **By Index** option.

OasReadNoChecksum

Value: boolean.

When set, the reader will ignore a checksum found in the OASIS file, if any. When not set, if a checksum is found, it will be compared with a computed checksum, using the method (CRC or summation) indicated in the file, and the conversion will fail if the checksums are not equal.

OasPrintNoWrap

Value: boolean.

This applies when converting OASIS input to ASCII text. When set, the text output for a single record will occupy one (arbitrarily long) line. When not set, lines are broken and continued with indentation.

This variable has a corresponding check box in the **ASCII Text** output format page of the **Format Conversion** panel.

OasPrintOffset

Value: boolean.

This applies when converting OASIS input to ASCII text. When set, the first token for each record output gives the offset in the file or containing CBLOCK. When not set, file offsets are not printed.

This variable has a corresponding check box in the **ASCII Text** output format page of the **Format Conversion** panel.

E.21 Convert Menu — Output

The **!set** variables below affect the format conversion when writing data to a file. Many of these variables have counterpart buttons in the **Export Control** panel from the **Convert Menu**. The functionality may also apply to files created with the **Save** command and similar.

The following table identifies where the variables in this section are set, if settable from the graphical interface, and specifies the scope of the variables.

Variable	Set From	Notes
StripForExport	**Format Conversion** and **Export Control**	4
WriteMacroProps	**Export Control**	1
KeepLibMasters	**Export Control**	3
SkipInvisible	**Export Control**	3
KeepBadArchive		1
NoCompressContext		5
RefCellAutoRename		5
UseCellTab		5
SkipOverrideCells		5
OutToLower	cell name mapping module	2
OutToUpper	cell name mapping module	2
OutUseAlias	cell name mapping module	2
OutCellNamePrefix	cell name mapping module	2
OutCellNameSuffix	cell name mapping module	2
CifOutStyle	**Export Control**	1
CifOutExtensions	**Export Control**	1
CifAddBBox		1
GdsOutLevel	**Export Control**	1
GdsMunit	**Export Control**	1
NoGdsMapOk	**Export Control**	1
OasWriteCompressed	**Export Control**	1
OasWriteNameTab	**Export Control**	1
OasWriteRep	**Export Control**	1
OasWriteChecksum	**Export Control**	1
OasWriteNoTrapezoids	**Advanced OASIS Export Parameters**	1
OasWriteWireToBox	**Advanced OASIS Export Parameters**	1
OasWriteRndWireToPoly	**Advanced OASIS Export Parameters**	1
OasWriteNoGCDcheck	**Advanced OASIS Export Parameters**	1
OasWriteUseFastSort	**Advanced OASIS Export Parameters**	1
OasWritePrptyMask	**Advanced OASIS Export Parameters**	1

Notes:

1. These variables apply whenever a layout file is being written, in any mode.

2. These variables apply to actions initiated from a panel containing the **Cell Name Mapping** control group, and to the following script functions:

```
ToXIC
ToCGX
ToCIF
ToGDS
ToGdsLibrary
ToOASIS
```

3. Applies when a file is being written using the **Export Control** panel, and with the script functions listed above.

4. The StripForExport variable applies as described below.

5. These variables apply when using a Cell Hierarchy Digest (CHD) to access cells for writing. Reference cells are pointers to CHD data.

StripForExport

Value: boolean.

When this variable is set, files produced through the **Export Control** and **Format Conversion** panels will contain the basic syntax elements with no extensions. Thus, they contain physical data only. The StripForExport variable applies when writing all output, **except** when using the **Save** and **Save As** buttons in the **File Menu**, and the equivalent text accelerators and including the prompts when exiting the program. It is also ignored when using the `Save` script function, but applies in the `ToArchive` script function.

Within *Xic*, archive file representations consist of two sequential records in each file. The first record is the physical information, and the second record contains the electrical information. These files should be compatible with other CAD systems, as these files are generally expected to have only one record, and consequently the electrical information may be ignored. However, one should not count on this. When in effect, only the physical record is output. This produces a file which should be an absolutely conventional physical layout file.

Additionally, when StripForExport is set, and when writing out a hierarchy from the main database, all cells in the hierarchy will be written, whether or not the KeepLibMasters variable is set. Thus, the file will not contain unsatisfied cell references, as (physical) library cells will be included. Further, all referenced pcell and standard via sub-masters will be written to output, similar to the case when the PCellKeepSubMasters and ViaKeepSubMasters variables are set.

This variable tracks the state of the **Strip For Export - (convert physical data only)** check box which appears in the **Export Control** and **Format Conversion** panels. This button should be active when creating a file to be sent to a vendor for use in generating photomasks. Note that the electrical information can never be recovered from a stripped file.

WriteMacroProps

Value: boolean.

When set, output will include macro properties, which are no longer in use in 4.3.6 and later. This variable can be set to force generation of these properties, thus providing backwards compatibility.

KeepLibMasters

Value: boolean.

When writing an archive file from a hierarchy in the main database, cells in the hierarchy that were opened through the library mechanism are by default **not** included in the file. References to these cells remain, though no library cell definition records will appear in output. The file will not be self-contained, as the library cell references are unresolved without the corresponding libraries.

When this variable is set, files produced with the **Export Control** panel will include all cells in the hierarchy, and the file produced will not have any unsatisfied references (except for electrical device library cells, which are never included in output). The variable also applies to the script functions listed in the notes to the table at the top of this section. It does *not* apply to the **Save** and **Save As** commands, which always omit library cells.

This variable tracks the state of the **Include Library Cells** check box in the **Export Control** panel.

SkipInvisible

Value: boolean or string.

When this variable is set, only layers that are currently visible, as selected with button 2 in the layer table or otherwise, will be converted when writing output from the **Export Control** panel. If set to a word beginning with '`p`' (case insensitive), only invisible physical layers will be skipped. If set to a word beginning with '`e`' (case insensitive) only the invisible electrical layers will be skipped. If set to anything else, including the empty string, both physical and electrical invisible layers will be skipped. This variable tracks the state of the **Don't convert invisible layers** check boxes in the **Export Control** panel.

NoCompressContext

Value: boolean.

The Cell Hierarchy Digest (CHD) is a data structure which provides a compact representation of a cell hierarchy found in an archive file. This data structure is used in operations where random-access of cells in the archive file is required. This is used in some of the conversion functions provided in the **Format Conversion** panel from the **Convert Menu**, and elsewhere.

In order to process large files, it is important that the CHD use as little memory as possible. In release 2.5.67 and later, a mechanism is used to compress instance lists by default. This can shrink the memory used by the CHD by 50computational overhead.

The digest files written by the **Save** button in the **Cell Hierarchy Digests** panel and the `WriteCellHierDigest` script function use the compressed instance lists by default, and are typically more compact than the older format. These files have a new magic number and can not be read by *Xic* releases prior to 2.5.67.

This boolean variable, if set, will prevent use of compression in the CHD structures, and files written will be backwards compatible. It is unlikely that the user will find it necessary to set this variable.

RefCellAutoRename

Value: boolean.

This variable applies when writing hierarchies containing reference cells, which are cells which point to data obtained through a Cell Hierarchy Digest but are otherwise empty. When written to a layout file, these cells expand into a full cell hierarchy obtained from the CHD. The output file can not contain more than one cell definition for a given name, so by default if a duplicate cell name is encountered when writing, that cell definition is simply skipped, and all instances of the cell in output will reference the original definition.

This is the correct thing to do when duplicate cell names come from the same (or an equivalent) CHD, as the duplicates really do indicate the same cell. However, if the names come from different CHDs, this could indicate a true name clash.

When this variable is set, names that clash, and that come from non-equivalent CHDs, will cause an automatic renaming of the cell, and a cell definition will be generated in output under the new name. The subsequent cell instances will be updated to call the new name. Names that clash but come from equivalent CHDs will have the cell definition skipped, as in the default mode.

This variable tracks the **Use auto-rename when writing CHD reference cells** check box in the **Cell Hierarchy Digests** panel from the ¡b¿File Menu¡/b¿.

UseCellTab
Value: boolean.
This variable enables cell definition substitution when using a Cell Hierarchy Digest (CHD) to access cells for purposes other than reading into main memory. When set, cell names found in the **Cell Table Listing**, which also are visible in the main database will replace cells of the same name when accessing a hierarchy through a CHD. This feature can be used to modify cells in a hierarchy without having to read the entire hierarchy into main memory.

This variable tracks the state of the **Use cell table** check box in the **Cell Hierarchy Digests** panel.

SkipOverrideCells
Value: boolean.
This variable applies only when UseCellTab is set. When this variable is also set, cell names listed in the **Cell Table Listing** will be skipped, rather than substituted. When writing output, this will produce files that have unresolved references, which can be satisfied by another source, such as a library.

This variable tracks the state of the **Override** and **Skip** radio buttons in the **Cell Table Listing** panel.

OutAllCells
Value: boolean.
When set, all cells in the current symbol table, not just the hierarchy of the current cell, will be output as if they were part of the hierarchy. The usual filtering of library and sub-master cells is retained. The resulting file may have multiple top-level cells. This variable tracks the state of the **Consider ALL cells in current symbol table for output** check box in the **Export Control** panel from the **Convert** main menu **Export Cell Data** button.

Out32nodes
Value: boolean.
When set, schematic cell data written to files will use the node property syntax of the 3.2 branch of *Xic*, providing limited backward compatibility. This will strip out elements not supported by the earlier syntax, such as multi-contact points in symbols.

The files will still not really be backward compatible unless all "new" features are avoided. Setting this variable may be useful for the case where 3.2 compatibility is to be preserved for a design that originated in 3.2 or earlier, which is read into the current release of *Xic*, tweaked, then saved back to disk.

The variable should not be set unless you explicitly need to create backward-compatible files, as it will prevent features from working in the resulting files.

OutToLower
Value: boolean.
When set, cell names found in archive files being written that are entirely upper case will be mapped to lower case. A name that is mixed-case will not be changed. This mapping occurs for names which are not aliased in an enabled alias file. This is part of a more general cell name mapping facility (see 14.2), which applies in the **Export Control** panel and elsewhere.

OutToUpper

Value: boolean.

When set, cell names found in archive files being written that are entirely lower case will be mapped to upper case. A name that is mixed-case will not be changed. This mapping occurs for names which are not aliased in an enabled alias file. This is part of a more general cell name mapping facility (see 14.2), which applies in the **Export Control** panel and elsewhere.

OutUseAlias

Value: boolean or string.

This variable enables utilization of the alias file (see 14.3) when writing to an archive file. If simply set as a boolean, i.e., to no value, the alias file will be read before the operation, and created or updated if necessary after the operation. If the variable is set to a word starting with '`r`' (case insensitive), then the alias file will be read before the operation and used during the operation (if it exists), but will not be created or updated after the operation. If the variable is set to a word starting with '`w`' or '`s`' (case insensitive), the alias file will not be read before an operation, but will be created or updated after the operation completes. This is part of a more general cell name mapping facility (see 14.2), which applies in the **Export Control** panel and elsewhere.

OutCellNamePrefix, OutCellNameSuffix

Value: string.

These variables are most simply set to a text token that is added to the beginning or end of cell name strings as archive files are being written. Modifications will not be made to cell names found in an enabled alias file. The strings can also be given in the form

/*str*/*sub*/

where *str* and *sub* are text tokens, separated by forward slash characters as shown. In this case if the characters at the beginning/end of the cell name (for prefix/suffix) match the *str*, they are replaced by *sub*. This is the same action as is used in the **!rename** command. The string token must match exactly — there is no wildcarding. Either the prefix or suffix, or both, can be defined. The suffix substitution occurs after the prefix substitution. Either can match the whole cell name if one wants to change the name of a single cell. This is part of a more general cell name mapping facility (see 14.2), which applies in the **Export Control** panel and elsewhere.

CIFoutStyle

Value: string.

When set, this variable will determine the CIF output style. Changing the **Cell Name Extension**, **Layer Specification**, or **Label Extension** option menu choices in the **CIF** page of the **Export Control** pop-up will update the value of CifOutStyle.

The CIFoutStyle variable can be set to the following values, which will set the CIF output style as indicated. The syntax associated with the indices is given in 14.7.3, describing the **Export Control** panel.

Value	Historical Name	cname_index	layer_index	label_index
`a`	Stanford	1	0	1
`b`	NCA	1	1	2
`i`	Icarus	2	0	1
`m`	Mextra	0	0	3
`n`	none	4	0	4
`s`	Sif	3	0	1
`x`	Xic	0	0	0
cn:*la*:*lb*	-	*cn*	*la*	*lb*

The final form consists of three colon-separated integers which are interpreted as indices into the option lists as implied above. If the style parameters are changed in the **Export Control** pop-up while CIFoutStyle is set, the value of CIFoutStyle will have this form.

CifOutExtensions

Value: two space-separated integers.

The string for this variable consists of two integers that represent banks of flags. The first integer represents the extension flags in use when the StripForExport variable is not set, the second integer represents the flags in force when StripForExport is set. The bits of each integer represent the flag state corresponding to the menu entries of the **CIF Extensions** menu (below the separator) in the **CIF** page of the **Export Control** panel, with the top entry corresponding to the least significant bit. The extensions are described with the CIF Format Extensions in /refcifext, and are listed in the table below.

Extension	**Mask**
scale extension	`0x1`
cell properties	`0x2`
inst name comment	`0x4`
inst name extension	`0x8`
inst magn extension	`0x10`
inst array extension	`0x20`
inst bound extension	`0x40`
inst properties	`0x80`
obj properties	`0x100`
wire extension	`0x200`
wire extension new	`0x400`
text extension	`0x800`

CifAddBBox

Value: boolean.

When set, each object line (boxes, polygons, wires, labels) in CIF output will be followed by a comment line giving the bounding box of the object, in the form

`(BBox` *left bottom right top*`);`

This may be useful for debugging, but greatly increases file size so is not recommended for general use.

In *Xic* releases prior to 3.0.0, the format of the added comment was

(BBox *left*,*top width height*);

and the extension was applied to native cell files as well as CIF output.

GdsOutLevel

Value: integer 0–2.

This variable determines the GDSII release level of GDSII output files. The default is release level 7, which was introduced by Cadence in 2002. Previous releases specified a limit of 200 or 600 polygon vertices (there seems to be some inconsistency in the published limit) and 200 vertices for wires. This applies to format releases 3, 4, 5, and 6. The only difference between these formats is the definition of some Cadence-specific data block types that are ignored by *Xic*. The latest release (7) removed these limits. The limits that remain are due to the block size limit (64Kb) of the format, which implies a maximum of 8000 vertices for polygons and wires.

When writing GDSII output, it may be necessary to enforce the limits, if the output is destined for another program which can't handle the release 7 limits. The *Xic* default is to use the release 7 limits.

The GdsOutLevel variable can be set to an integer 0–2. The corresponding GDSII format is as follows:

level 0: (the default)
max poly vertices: 8000
max wire vertices: 8000
format level: 7

level 1:
max poly vertices: 600
max wire vertices: 200
format level: 3

level 2:
max poly vertices: 200
max wire vertices: 200
format level: 3

By setting GdsOutLevel to 1 or 2, GDSII files generated with *Xic* should not cause difficulty when read by older programs (including old versions of *Xic*).

This variable tracks the state of the **GDSII version number, polygon/wire vertex limit** menu in the **GDSII** page of the **Export Control** panel from the **Convert Menu**. This page is also used in the **Format Conversion** panel, and the **Layout File Merge Tool** also from the **Convert Menu**.

GdsMunit

Value: real 0.01–100.0.
When writing GDSII, the normal MUNITS (machine units) and UUNITS (user units) values will be multiplied by this factor, and all coordinates in the file will be divided by this factor. The acceptable range is 0.01 – 100.0. This will apply to *all* GDSII files written.

This variable tracks the **Unit Scale** entry in the **GDSII** page of the **Export Control** panel from the **Convert Menu**. This page is also used in the **Format Conversion** panel, and the **Layout File Merge Tool** also from the **Convert Menu**.

The default values for these parameters are

MUNITS: 1e-6/*resolution*
UUNITS: 1.0/*resolution*

where *resolution* is the internal resolution, which defaults to 1000 per-micron, but can be changed with the DatabaseResolution variable.

GdsTruncateLongStrings

Value: boolean.
The GDSII and CGX formats use a 16-bit integer to store record size, limiting the size of records to 64Kb. This prevents storage of stings longer than this. By default, an attempt to write such a string to a GDSII or CGX file will generate a fatal error, aborting the operation. If this variable is set, overrunning strings will be truncated to maximum possible length, and the operation will continue without error. Warnings will appear in the log file, however.

This variable tracks the state of the **Accept but truncate too-long strings** check box in the **GDSII** and **CGX** pages of the **Export Control** panel from the **Convert Menu**. These pages are also used in the **Format Conversion** panel, and the **Layout File Merge Tool** also from the **Convert Menu**.

NoGdsMapOk

Value: boolean.

When this variable is set, layers without a GDSII output mapping will be ignored when producing GDSII output, though a warning will appear in the log file. Otherwise, this is an error which terminates conversion.

This tracks the state of the **Skip layers without Xic to GDSII layer mapping** check box in the **GDSII** and **OASIS** pages of the **Export Control** panel from the **Convert Menu**. These pages are also used in the **Format Conversion** panel, and the **Layout File Merge Tool** also from the **Convert Menu**.

OasWriteCompressed

Value: boolean, or the string "`force`".

When set, created OASIS files will use compression. The content of all CELL records and name tables will be placed in CBLOCK records. This can significantly reduce file size. When not set, no compression will be used.

By default, very short records are not compressed, as more often than not, compression will *increase* the size of these blocks. If this variable is set to the word "`force`", then all blocks are compressed. This can be used for comparison purposes, but is unlikely to yield the best results. This tracks the state of the check box in the **OASIS** page of the **Export Control** panel.

OasWriteNameTab

Value: boolean.

When set, all strings including cell names, properties, and labels are placed in strict-mode tables. This will in most cases reduce file size. When writing OASIS files with StripForExport set, i.e., writing physical data only, the offset table is placed in the END record. With StripForExport not set, in general we write two sequential OASIS databases into the file, the first for physical data, the second for electrical. In this case, string tables are used in the physical part only, and the offset table is placed in the START record. PAD records are added to avoid overwriting data since this is a non-sequential operation. In all cases, strict-mode tables are used.

The string tables themselves are written just ahead of the END record in all cases (when tables are used).

This tracks the state of the check box in the **OASIS** page of the **Export Control** panel.

OasWriteRep

Value: string or boolean.

When this variable is set, *Xic* will try to find groups of identical objects that can be combined into REPETITION records in OASIS output. This applies to all OASIS output files. Although compute intensive, this can save a lot of space in the output file.

If OasWriteRep is not set, subcell and object records are written as encountered when traversing the cell structure. If set, objects and subcells will be cached, and similar objects and subcells are identified and written using repetition records.

When using repetition, the following procedure is used, where "objects" can apply to subcells as well as geometrical objects.

1. Instead of directly converting each object, the object is saved in a cache.
2. When a cell traversal is complete or an object count reached, the cache is processed, and objects that are identical are identified. The differing objects are sorted to make use of modal variables.
3. For each group of identical objects, those that form a spatially linear, periodic "run" are extracted into a new run list.

4. For each list of runs, the runs that are spatially periodic are extracted into a new array list.
5. Each array is written using a 2-dimensional repetition.
6. Each remaining run is written using a 1-dimensional repetition.
7. The remaining objects, i.e., those not used in an array or run, are written using a random repetition.

The details of this process, and whether or not it is applied, are controlled by the OasWriteRep variable. This variable can be set to a string containing several tokens, or set as a boolean (i.e., set to nothing). The tokens can appear in any order.

OasWriteRep: [*word*] [d] [r] [m=N] [a=N] [x=N] [t=N]

word
: This is a token that is not recognized as one of the others. It consists of letters that control the type of object that the replication process is applied to. If the letter is present, the corresponding object type will be processed, otherwise the replication algorithm will not be applied to that type of object, however if this token is not found (no letters appear), all objects will be processed. The letters are:

c	subcells
b	boxes
p	polygons
w	wires
l	labels

For example, "cp" would indicate use of replications for subcells and polygons only. If no token of this type is found, then *all* object types will be processed.

The remaining tokens are identified by the first letter only, and the remainder of the token (up to '=' in some cases) is ignored.

d
: Some debugging info is printed on the console when processing.

r
: No attempt is made to find runs or arrays, and all similar objects are written using random placement repetitions.

m=N
: This sets the minimum number of objects in a run. The default value is 4, which is also the minimum accepted value. There can be no space around the '=', and N must be an integer. This is ignored if r is given.

a=N
: This sets the minimum number of runs in an array. The default value is 2. The value can be set to 0 (zero) in which case two dimensional repetition finding is skipped. Otherwise, the value must be 2 or larger. There can be no space around the '=', and N must be an integer. This is ignored if r is given.

x=N
: This sets the maximum number of different objects of a given type held in the cache, before flushing occurs. This does not include repetition counts. The N is an integer in the range 20 – 50000. If not set, a default of 5000 is used. Larger values can reduce file size, but can greatly increase writing time due to modality sorting.

t=N

This sets the maximum number of similar objects, i.e., those subject to repetition analysis, that can exist in the cache before flushing. Extremely large numbers may require excessive time to scan for repetitions. The N is an integer which can be 0 (zero) in which case no limit is used, or 100 or larger. The default value is 1000000 (one million).

If OasWriteRep is set to an empty string, all objects will be processed for replication, using the default run and array minimums.

The string for this variable can be composed with the interface found in the **Advanced OASIS Export Parameters** panel. The **Find repetitions** button in the **OASIS** page of the **Export Control** panel will set the variable to the current string from the interface, or unset the variable. It the variable is set by another method, such as with the **!set** command, the interface will be updated to the parameters as given. With default parameters, the string is empty, so the variable is set as a boolean by default.

OasWriteChecksum

Value: string or boolean.

When not set, no checksum is written to the output. When set as a boolean (i.e., to no value), or to anything other than "2" or a string beginning with "ch", a cyclic-redundancy (CRC) checksum is computed and added to the file. If set to "2" or a word beginning with "ch", a byte-sum checksum is added to the file. This variable has a corresponding check box in the **OASIS** page of the **Export Control** panel. This controls setting/unsetting as a boolean, thus the check box selects CRC checksum or none.

OasWriteNoTrapezoids

Value: boolean.

The normal behavior is to check three and four-sided polygons to see if they can be written as (more compact) TRAPEZOID or CTRAPEZOID records. Setting this variable will suppress this, providing slightly faster conversion at the cost of larger file size. This variable tracks the **Don't write trapezoid records** check box in the **Advanced OASIS Export Parameters** panel.

OasWriteWireToBox

Value: boolean.

The normal behavior is to leave wires alone, preserving data-type integrity. However, space can be saved by writing two-vertex rectangular wires as boxes. Setting this variable will enable this, which may reduce file size at the expense of slightly more conversion time. This variable tracks the **Convert Wire to Box records when possible** check box in the **Advanced OASIS Export Parameters** panel.

OasWriteRndWireToPoly

Value: boolean.

The OASIS format does not have a native "rounded end" style for wires. These are normally converted to extended-end wires, where the "rounded" part becomes Manhattan. If this variable is set, when converting rounded-end wires to OASIS, the wire is converted to a polygon which is shaped the same way as all rounded-end wires in *Xic*. Use of a polygon requires more memory than the wire, but this preserves exactly the same geometrical coverage, which is valuable in reducing geometric differences if a layout comparison is performed. This variable tracks the **Convert rounded-end Wire records to Poly records** check box in the **Advanced OASIS Export Parameters** panel.

OasWriteNoGCDcheck

Value: boolean.

This applies only when repetitions are being used (OasWriteRep is set). Normally, a greatest common divisor is computed, and if larger than unity type 10 repetitions are converted to type 11. This can reduce file size. If this variable is set, the GCD is not computed, probably increasing file size but reducing conversion time. This variable tracks the **Skip GCD check** check box in the **Advanced OASIS Export Parameters** panel.

OasWriteUseFastSort
Value: boolean.
When set, writing OASIS may be faster at the expense of file size. This was the only mode in releases prior to 2.5.68. The present release defaults to using a somewhat slower but more effective modality sorting algorithm, which will produce smaller files. This variable tracks the **Use alternate modal sort algorithm** check box in the **Advanced OASIS Export Parameters** panel.

OasWritePrptyMask
Value: boolean or string.
This variable tracks the **Property masking** menu selections in the **Advanced OASIS Export Parameters** panel.

There are two properties that are added to text labels by default. These properties are used by *Xic* and programs based on *Xic* source code, and can be stripped if not needed. This can lead to substantial file size reduction if the file contains many text labels.

Property name: `XIC_PROPERTIES`
Property number: `7012`

This property is added when reading GDSII source. It contains values of attributes of the TEXT element. These have no analogs in OASIS format, however if the file is reconverted to GDSII, the attributes will be restored. These attributes are found in the following GDSII record types:

name	**record**	**description**
ANGLE	28	Rotation angle of text.
MAG	27	Magnification applied to text.
WIDTH	15	Width of path used to form characters.
PTYPE	33	GDSII PATHTYPE used to form characters.

The property consists of a string containing name/value pairs: the names are the text tokens above, the values are numeric. Tokens are separated by white space.

Property name: `XIC_LABEL`
This is added to all labels to pass the *Xic* presentation attributes. The string consists of two space-separated unsigned numbers: *width* and *flags*. The *width* is the width of the label bounding box, in containing-cell coordinates. The *flags* is the label flags word used by *Xic*, described in C.2.

If OasWritePrptyMask is set as a boolean, i.e., to an empty string, neither of these properties is written. If the variable is set to an integer value, the two least-significant bits of the integer value are flags that mask the creation of these properties, according to the table below. If the variable is set to a non-empty and non-integer value, and during conversions only (as initiated from the **Format Conversion** panel from the **Convert Menu**) then *all* properties are stripped from output.

Bit 0: If set, `XIC_PROPERTIES` #7012 will not be written.
Bit 1: If set, `XIC_LABEL` will not be written.

This variable was named "OasWriteNoXicTextPrps" in releases prior to 3.0.0.

E.22 Custom Property Filtering

The **!set** variables below save property filter specification strings (see 14.13.3) for use when comparing layout data. The **!compare** command and the **Compare Layouts** panel available from the **Convert** menu provide this comparison function. The strings are used when the custom property filtering option is enabled.

PhysPrpFltCell
Value: string.
Contains the custom filter string for physical cell properties.

PhysPrpFltInst
Value: string.
Contains the custom filter string for physical instance properties.

PhysPrpFltObj
Value: string.
Contains the custom filter string for physical object properties.

ElecPrpFltCell
Value: string.
Contains the custom filter string for electrical cell properties.

ElecPrpFltInst
Value: string.
Contains the custom filter string for electrical instance properties.

ElecPrpFltObj
Value: string.
Contains the custom filter string for electrical object properties.

E.23 Design Rule Checking

These variables are used by the design rule checking (DRC) system and are not generated by or recognized in the *XicII* or *Xiv* feature sets. Unless stated otherwise, these settings can be controlled from the **DRC Defaults** panel from the **Set Defaults** button in the **DRC Menu**.

Drc
Value: boolean.
This sets whether or not the interactive rule checking is applied to objects being added to the database, tracking the state of the **Enable Interactive** button in the **DRC Menu**.

DrcNoPopup
Value: boolean.
This variable determines whether errors generated in interactive DRC will be listed in a pop-up window. If set, the messages will not pop up automatically. This initializes the state of the **No Pop Up Errors** button in the **DRC Menu**.

DrcLevel
Value: integer 0–2.
This sets the error recording level for design rule checking. If set to zero (“0”) or not set, only one

violation is recorded per object. If 1, one violation of each type is recorded per object. If 2, all violations found are recorded.

`DrcMaxErrors`

Value: integer 0–100000.
This variable sets the maximum number of design rule violations reported in batch mode, at which point checking terminates. If set to zero or not set, no limit is imposed.

DrcInterMaxObjs

Value: integer 0–100000.
In interactive design rule checking, this variable provides a limit on the number of objects checked, to minimize the pause after an operation. If set to 0, no limit is imposed. If not set, a limit of 1000 is taken.

DrcInterMaxTime

Value: integer 0–30000.
This variable limits the time of the interactive design rule checking performed after each operation. The value is given in milliseconds. If the value is 0, there is no time limit imposed. If the variable is not set a limit of 5000 (five seconds) is assumed.

DrcInterMaxErrors

Value: integer.
This variable limits the number of violations to record during interactive testing. When the limit is reached, testing stops and control returns to the user. If set to 0, there is no limit. If not set, a limit of 100 violations is imposed.

DrcInterSkipInst

Value: boolean.
If a subcell is copied, moved, or placed, by default the subcell is tested for design rule violations if in interactive mode. Setting this variable will cause this checking to be skipped. The checking may be redundant and time consuming.

DrcChdName

Value: string.
It is possible to use a Cell Hierarchy Digest (CHD) to specify a target layout for design rule checking. This can allow DRC testing of layouts that are too large to be read into *Xic* normally. This value mirrors the contents of the **CHD reference name** text entry area in the **DRC Run Control** panel from the **Batch Check** button in the **DRC Menu**.

DrcChdCell

Value: string.
This variable stores an optional cell name for use as the top-level cell when a CHD is used for DRC. It mirrors the contents of the **CHD top cell** text entry area in the **DRC Run Control** panel from the **Batch Check** button in the **DRC Menu**.

DrcLayerList

Value: string.
It is possible to use only rules on certain layers, or to skip rules on certain layers, when running DRC. This variable contains a space separated list of layer names for use in the layer filtering. It mirrors the contents of the **Layer List** text entry area in the **DRC Parameter Setup** panel from the **Setup** button in the **DRC Menu**.

DrcUseLayerList

Value: boolean or string.

If this variable is set to a word that starts with 'n' (case insensitive) the layers listed in the DrcLayerList variable will be skipped during DRC runs, meaning that the rules defined on the skipped layers will not be evaluated. If DrcUseLayerList is set to anything else, including to an empty string (i.e., as a boolean), then only rules on layers listed in the DrcLayerList variable will be checked during DRC runs. In this case, if the DrcLayerList is not set or empty, the filtering is not done, and rules on all layers will be checked. This variable sets, and is set by, the **Check listed layers only** and **Skip listed layers** check boxes in the **DRC Parameter Setup** panel from the **Setup** button in the **DRC Menu**.

DrcRuleList
Value: string.
It is possible to use only certain rules, or to skip certain rules, when running DRC. This variable contains a space separated list of rule names (technology file rule keywords) for use in this filtering. It mirrors the contents of the **Rule List** text entry area in the **DRC Parameter Setup** panel from the **Setup** button in the **DRC Menu**. Rule name matching is case-insensitive.

DrcUseRuleList
Value: boolean or string.
If this variable is set to a word that starts with 'n' (case insensitive) the rules listed in the DrcRuleList variable will be skipped during DRC runs. If DrcUseRuleList is set to anything else, including to an empty string (i.e., as a boolean), then only rules listed in the DrcRuleList variable will be checked during DRC runs. In this case, if the DrcRuleList is not set or empty, the filtering is not done, and all rules will be checked. This variable sets, and is set by, the **Check listed rules only** and **Skip listed rules** check boxes in the **DRC Parameter Setup** panel from the **Setup** button in the **DRC Menu**.

DrcPartitionSize
Value: real number.
When this variable is set to a real number larger than `0.0`, batch mode DRC initiated from the **DRC Run Control** panel will use a square grid of the indicated size in microns. The DRC tests will be performed sequentially in each of the grid areas that overlap the overall test area. This variable mirrors the state of the **Partition grisd size** entry area and **None** button in the **DRC Run Control** panel.

E.24 Extraction Tech

These are mostly in support of the extraction system, but the variables and keywords are handled by the main program, so can be set or read if the extraction system is not available.

AntennaTotal
Value: real number.
This variable applies to the **!antenna** command. The value is a threshold total-net antenna ratio, as explained for the **!antenna** command. The value is effectively passed to that command as a default.

The **Global Attributes** button in the **Tech Parameter Editor** provides a prompt-line interface for setting this variable.

Db3ZoidLimit
Value: integer 1000 or larger.
This limits the amount of geometry which can be saved in the 3-D geometry database, which is

used in the **Cross Section** command, and in the interfaces to external capacitance and inductance extraction programs. The total trapezoid element count is limited to 10000 by default, i.e., when this variable is not set. The database is not designed for large collections, and the limit avoids embarking on long computations where the program becomes unresponsive.

LayerReorderMode
Value: integer 0–2.
This sets the default sequencing assumption used in the three-dimensional layer sequence generator (see 12.8), which is used for the cross-section display and the capacitance extraction interface. This can be set to an integer in the range 0–2. The value 0 is the default, the same as if the variable is not set. The other values will internally resequence `Via` layers, as described for the layer sequence generator.

The **Global Attributes** button in the **Tech Parameter Editor** provides a prompt-line interface for setting this variable.

NoPlanarize
Value: boolean.
If set, by default no layers are planarizing, as explained in the description of the three-dimensional layer geometry database in 12.8. Otherwise, the default is that layers with the `Conductor` keyword given, explicitly or implicitly, or the `Via` keyword given, will be planarizing by default. The `Routing`, `GroundPlane`, `GroundPlaneClear`, `Contact` and their aliases implicitly set the `Conductor` keyword. Thus, by default the metal stack is planarized, as in a contemporary semiconductor process.

The **Global Attributes** button in the **Tech Parameter Editor** provides a prompt-line interface for setting this variable.

SubstrateEps
Value: real number.
This variable sets the relative dielectric constant assumed for the substrate, used by the capacitance extraction interface. If not set, the default is 11.9.

The **Global Attributes** button in the **Tech Parameter Editor** provides a prompt-line interface for setting this variable.

SubstrateThickness
Value: real number.
This variable sets the thickness of the substrate assumed by the program, as a real number in microns. This is used only by the capacitance extraction interface. If not set, a thickness of 75.0 microns will be assumed.

The **Global Attributes** button in the **Tech Parameter Editor** provides a prompt-line interface for setting this variable.

E.25 Extraction General

The following variables control features of the general extraction and association process.

ExtractOpaque
Value: boolean.
When set, *Xic* will ignore the `OPAQUE` flag and perform extraction normally on cells with this flag set. The `OPAQUE` flag would otherwise suppress extraction on the contents of the cell. This flag is set in the flags property of physical cells.

This tracks the setting of the **Extract opaque cells, ignore OPAQUE flag** check box in the **Net and Cell Config** page of the **Extraction Setup** panel from the **Setup** button in the **Extract Menu**.

FlattenPrefix

Value: string.
This variable can be set to a string containing a space-separated list of words. The words are intended to match cell names or classes of cell names. Cells with names that match are **not** treated as individual cells during extraction, instead they are treated as if instantiations are part of the containing cell, i.e., they are logically flattened (see 16.4). This applies to physical cells only, and such cells will have no recognized electrical counterpart.

Note: it is probably more convenient to set the flatten property of physical cells that should be flattened into their parent during extraction. Setting this property with the **Cell Property Editor** will have the same effect as including the cell in the FlattenPrefix list, but is persistent when the cell is saved.

In the words, the forward slash character ('/') is special, and is used to indicate the type of matching. The possibilities are:

name[/]
This will prefix match cell names, the trailing '/' is optional. For example if *name* is "`abc`", cell names `abc`, `abc123`, and `abcounter` would match.

/*name*
This will suffix match cell names. For example, if the word is "`/bar`", cell names `bar`, `foobar`, and `crossbar` would match.

/*name*/
This will literally match a cell name, for example `/foobar/` would match only a cell named `foobar`.

This tracks the setting of the **Cell flattening name keys** entry in the **Net and Cell Config** page of the **Extraction Setup** panel, which is obtained from the **Setup** button in the **Extract Menu**.

Note: in *Xic* releases prior to 3.1.8, this variable could be set to a single word only, and prefix matching was always employed. In releases of *Xic* prior to 2.5.19, this variable was named "PnetFlattenPrefix".

GlobalExclude

Value: string (layer expression).
This variable can be set to a layer expression (which includes the case of a layer name). Any object in the layout which touches a region where the layer expression evaluates as dark will be ignored by the extraction system. This facilitates use of special layers to mask off parts of a layout to be ignored in extraction.

This tracks the setting of the **Global exclude layer expression** entry in the **Misc Config** page of the **Extraction Setup** panel, which is obtained from the **Setup** button in the **Extract Menu**.

GroundPlaneGlobal

Value: boolean.
When set, every object in every cell on a clear-field ground plane layer is assigned to group 0. If not set, only the largest area group on this layer, in the top-level cell, is assigned to group 0.

This tracks the setting of the **Assume clear-field ground plane is global** check box in the **Net and Cell Config** page of the **Extraction Setup** panel from the **Setup** button in the **Extract Menu**.

GroundPlaneMulti

Value: boolean.

When set, a layer specified as GroundPlaneClear in the technology file will be inverted, and the inverted version used for grouping and extraction. The MultiNet keyword which optionally follows GroundPlaneClear in the technology file effectively sets this variable. If this variable is unset, then no inversion takes place, and the absence of the GroundPlaneClear layer is taken to indicate ground (group 0). This variable has no effect unless a GroundPlaneClear layer exists.

Note: This replaces the `HandleTermDefault` variable which existed in earlier *Xic* releases. It is part of the ground plane support in the extraction system.

This tracks the setting of the **Invert dark-field ground plane for multi-nets** check box in the **Net and Cell Config** page of the **Extraction Setup** panel from the **Setup** button in the **Extract Menu**.

GroundPlaneMethod

Value: integer 0–2.

This sets the method used to invert the ground plane for grouping and extraction, if the MultiNet keyword has been applied to a GroundPlaneClear layer in the technology file. The possible values are integers 0–2, which have the same meaning as the integer that optionally follows MultiNet in the technology file (see A.6.4).

This tracks the setting of the inversion method menu in the **Net and Cell Config** page of the **Extraction Setup** panel from the **Setup** button in the **Extract Menu**.

KeepShortedDevs

Value: boolean.

By default, if an extracted device is found to have all terminals shorted together at the time the device is recognized, the device will be ignored. This will help reject spurious devices from test structures, etc.

If the KeepShortedDevs variable is set, then these devices will be kept (as in pre-2.5.69 releases). This flag may be needed for LVS to pass, if the schematic contains the shorted devices.

This tracks the setting of the **Include devices with terminals shorted** check box in the **Device Config** page of the **Extraction Setup** panel, which is obtained from the **Setup** button in the **Extract Menu**.

MaxAssocLoops

Value: integer 0–1000000.

This variable sets a parameter used by the association algorithm. Presently, it is not expected to be useful to the user, and it is recommended that it not be changed.

The variable tracks the setting of the **Maximum association loop count** entry in the **Misc Config** page of the **Extraction Setup** panel from the **Setup** button in the **Extraction Menu**.

MaxAssocIters

Value: integer 10–1000000.

This variable sets a parameter used by the association algorithm. Presently, it is not expected to be useful to the user, and it is recommended that it not be changed.

The variable tracks the setting of the **Maximum association iterations** entry in the **Misc Config** page of the **Extraction Setup** panel from the **Setup** button in the **Extraction Menu**.

NoMeasure

Value: boolean.

This turns off the extraction of parametric data for devices in the extraction system. This is mainly for debugging, but may save time if the user is interested in topology only. The measurements can be time consuming.

This tracks the setting of the **Skip device parameter measurement** check box in the **Device Config** page of the **Extraction Setup** panel from the **Setup** button in the **Extract Menu**.

UseMeasurePrpty
Value: boolean.
When set, the extraction system will read and update (creating if necessary) the measures property (property number 7106) which is used to cache (see 16.7) measurement results. The measurement of device parameters can be time consuming, and the caching can speed up the extraction process significantly. However, using the measurement cache may require user intervention to maintain coherency. If a device layout changes, the user will have to manually update the cache in order to obtain updated parameters. With this variable unset, the default condition will force actual computation of device parameters, and avoid all use of the caching mechanism. This is appropriate while a cell is under development, to avoid cache coherency issues.

This variable tracks the **Use measurement results cache property** check box in the **Device Config** page of the **Extraction Setup** panel from the **Setup** button in the **Extract Menu**.

NoReadMeasurePrpty
Value: boolean.
This variable is ignored unless UseMeasurePrpty is set. When set, the extraction system will not read the measures property (property number 7106) which is used to cache (see 16.7) measurement results. When measurement results are required, they will be computed. The property will still be updated, after association, if UseMeasurePrpty is set. Thus, by setting this variable and forcing association, one can get a fresh set of measurement results into the measures properties.

This variable tracks the **Don't read measurement results from property** check box in the **Device Config** page of the **Extraction Setup** panel from the **Setup** button in the **Extract Menu**.

NoMergeParallel
Value: boolean.
Setting this variable suppresses merging of parallel-connected devices during extraction. This applies to all devices, and supersedes the Merge directive in the device blocks or the technology file.

This variable tracks the setting of the **Don't merge parallel devices** check box in the **Device Config** page of the **Extraction Setup** panel, which is obtained from the **Setup** button in the **Extract Menu**.

NoMergeSeries
Value: boolean.
Setting this variable suppresses merging of series-connected devices during extraction. This applies to all devices, and supersedes the Merge directive in the device blocks of the technology file.

This variable tracks the setting of the **Don't merge series devices** check box in the **Device Config** page of the **Extraction Setup** panel, which is obtained from the **Setup** button in the **Extract Menu**.

NoMergeShorted
Value: boolean.
When including devices with all terminals shorted (the KeepShortedDevs variable is set), setting this variable will prevent such devices from being merged as parallel devices, if parallel merging is enabled for the device type.

This variable tracks the setting of the **Don't merge devices with terminals shorted** check box in the **Device Config** page of the **Extraction Setup** panel, which is obtained from the **Setup** button in the **Extract Menu**.

IgnoreNetLabels
Value: boolean.
If set, net name labels will be ignored by the extraction system. This is probably only useful for debugging. Although this may allow correct association if a net name label is wrong, the recommended solution is to correct the offending label.

This variable tracks the setting of the **Ignore net name labels** check box in the **Net and Cell Config** page of the **Extraction Setup** panel, which is obtained from the **Setup** button in the **Extract Menu**.

UpdateNetLabels
Value: boolean.
When set, net name labels will be updated, and new net name labels possibly created, after association completes. The label text is obtained from corresponding electrical net names.

This is a dangerous operating mode, as if association fails, it is possible that incorrect net name labels will be created. These will subsequently prevent correct association and cause LVS failure, until removed or corrected by hand.

When creating library cells, running extraction with this variable set can be a final action before saving the finished cell. This must only be done if the cell passes LVS. The created net name labels should improve association efficiency, but are not essential.

This variable tracks the state of the **Update net name labels after association** check box in the **Net and Cell Config** page of the **Extraction Setup** panel, which is obtained from the **Setup** button in the **Extract Menu**.

FindOldTermLabels
Value: boolean.
When this variable is defined, *Xic* will recognize the "term labels" of earlier releases as net labels. In *Xic*-3, term labels were used (optionally) to specify the conductor groups that were associated with cell terminals in layouts. These are labels, created by the user on conducting layers, placed over an object on the same layer.

The term labels would also be recognized as net labels if the PinPurpose variable is set to an empty string, or the "`drawing`" purpose name. Setting the FindOldTermLabels is redundant in that case. The label searches are separate, and both will be done if enabled.

Whether this variable is set or not mirrors the status of the **Find old-style net (term name) labels** check box in the **Net and Cell Config** page of the **Extraction Setup** panel from the **Extract Menu**.

MergeMatchingNamed
Value: boolean.
If two physically unconnected conductor groups have the same logical net name (see 16.5), if this variable is set the groups will be logically merged and treated as a single group. This allows successful top-level LVS of cells containing split nets. Below the top level, split nets are detected by other means so setting this variable is not required for successful LVS if the top-level cell contains no split nets.

The group names that apply are obtained from net name labels, or from cell terminals that have been placed by the user. By default, net name matching is case-insensitive, though this can be changed with the NetNamesCaseSens variable. The name matching also treats as equivalent various subscripting delimiters, as listed in the description of the Subscripting variable.

This variable tracks the state of the **Merge groups with matching net names** check box in the **Net and Cell Config** page of the **Extraction Setup** panel, which is obtained from the **Setup** button in the **Extract Menu**.

MergePhysContacts

Value: boolean.
When set, additional association logic is employed to detect and account for split nets in instance placements. A "split net" is a logical net consisting of two or more disjoint physical conductor groups. The disjoint parts of the net are connected when instances are placed, through parent cell metalization. If the schematic shows the net fully connected in the master, LVS will fail on the parent unless this variables is set.

This variable tracks the state of the **Logically merge physical contacts for split net handling** check box in the **Misc Config** page of the **Extraction Setup** panel, which is obtained from the **Setup** button in the **Extract Menu**.

NoPermute

Value: boolean.
When this variable is set, the association algorithm will not attempt to use symmetry trials to find a solution. Symmetry trials are normally used to iterate through permutations when searching for a solution. During a trial, a particular set of associations is assumed, and the algorithm continues. If an inconsistency is found later, the associations made during the trial are reverted, and a new trial is started.

Many circuits do not require a permutation search. In some circuits, though, the permutation search can be a very time-consuming process. In circuits where association is known to fail perhaps because the wiring is incomplete, setting this variable will save time. This variable is mostly for debugging, or for cases where association is not needed. Of course, if a permutation search is needed and not performed, LVS will fail.

Permutes are also skipped if a device or subcircuit is found that can not possibly be associated.

This tracks the setting of the **Don't run symmetry trials in association** check box in the **Misc Config** page of the **Extraction Setup** panel, obtained from the **Setup** button in the **Extract Menu**.

PinLayer

Value: string.
If this variable is set to a layer name (or layer-purpose pair name) all net name labels must appear on the named layer. The "`pin`" purpose, and any setting of the inPurpose variable, are ignored.

The label will be associated with the conducting object containing the label origin that is highest (farthest from the substrate) in the layer table. Possible ambiguity with the associated layer makes this scheme not recommended, but support is present for compatibility with older cell libraries, such as the open-source CMOS libraries from Oklahoma State University.

This variable tracks the **Net label layer** entry in the **Net Config** page of the **Extraction Setup** panel, obtained form the **Setup** button in the **Extract Menu**.

PinPurpose

Value: string.
This applies when the PinLayer variable is not set. By default, net name labels must reside on a layer-purpose pair where the purpose name is "`pin`". However, if this variable is set to another valid purpose name, then that name will be required of net labels instead.

If the property is set to an empty string (i.e., as a boolean), the "`drawing` purpose is assumed. One could equivalently give the name explicitly. This is not really recommended as it can be inefficient.

This variable tracks the **Net label purpose name** entry in the **Net Config** page of the **Extraction Setup** panel, obtained form the **Setup** button in the **Extract Menu**.

RLSolverDelta

Value: floating point >= 0.01.
It this value is set, the resistance/inductance extractor will assume this grid spacing, in microns. The number of grid cells enclosed in the device will increase for physically larger devices, so that larger devices will take longer to extract. If this variable is set, the other RLSolver variables are ignored. Setting this variable may be appropriate if all resistors are "small" and dimensions conform to a layout grid.

This tracks the setting of the **Set/use fixed grid size** entry in the **Device Config** page of the **Extraction Setup** panel, which is obtained from the **Setup** button in the **Extract Menu**.

RLSolverTryTile

Value: boolean.
If set, the extractor will attempt to use a grid that will fall on every edge of the device body and contacts. The device and contact areas must be Manhattan for this to work. If such a grid can be found, and the number of grid cells is a reasonable number, this will give the most accurate result.

This tracks the setting of the **Try to tile** check box in the **Device Config** page of the **Extraction Setup** panel, which is obtained from the **Setup** button in the **Extract Menu**.

RLSolverGridPoints

Value: integer 10–100000.
When not tiling (RLSolverTryTile is not set), this sets the number of grid points used for resistance/inductance extraction. This number will be the same for all device structures, so that computation time per device is nearly constant. Higher numbers give better accuracy but take longer. The value used if not set is 1000.

This tracks the setting of the **Set fixed per-device grid cell count** entry in the **Device Config** page of the **Extraction Setup** panel, which is obtained from the **Setup** button in the **Extract Menu**.

RLSolverMaxPoints

Value: integer 1000–100000.
When tiling (RLSolverTryTile is set), the maximum number of grid cells is limited to this value. If the tile is too small, it will be increased in size to keep the count below this value, in which case the tiling will not have succeeded so there may be a small loss of accuracy. Using a large number of grid points can take a long time. The value used if not set is 50,000.

This tracks the setting of the **Maximum tile count per device** entry in the **Device Config** page in the **Extraction Setup** panel, which is obtained from the **Setup** button in the **Extract Menu**.

SubcPermutationFix

Value: boolean.
Setting this variable enables additional association logic. It applies when there is perfect topological matching between layout and schematic, but LVS is failing due to different permutations of permutable subcell contacts being assumed in the electrical and physical parts. Setting the variable will enforce the electrical permutation on the physical solution, which will allow LVS to pass if the permutation difference was the only issue.

This should no longer be needed, as the two-pass association algorithm in current use should resolve these cases automatically. This variable should therefor not be set in general, but it is possible that it might allow successful LVS in some obscure case.

This variable tracks the **Apply post-association permutation fix** check box in the **Misc Config** page of the **Extraction Setup** panel, which is obtained form the **Setup** button in the **Extract menu**.

VerbosePromptline
Value: boolean.
When set, lots of messages will be printed on the prompt line during extraction. Otherwise not much is printed, which may speed things up. This variable is linked to the **Be very verbose on prompt line during extraction** check box of the **Misc Config** page of the **Extraction Setup** panel.

ViaCheckBtwnSubs
Value: boolean.
By default, it is assumed that connections between subcells will be made by touching metal only. This includes the case where the metal is from a flattened wire-only cell, as would be provided by via cells as described in 16.9.2. One can easily adapt layout methodology where this is true. Otherwise, this variable can be set, which will cause explicit testing for the presence of vias between subcircuit nets. This is a very expensive operation.

Whether this variable is set or not tracks the state of the **Check for via connections between subcells** check box in the **Net Config** page of the **Extraction Setup** panel from the **Extract Menu**.

ViaSearchDepth
Value: non-negative integer.
If we have intersecting areas of top and bottom conductor, and we are searching for an area of via material that would connect the two metal objects, this sets the depth in the current cell hierarchy to search (see 16.9.2). The default is zero, indicating to search the current cell only. Generally, layout methodology can easily ensure that this value can be safely zero, but there may be cases that require extraction where such methodology was not practiced. In such a case, where the methodology is completely unknown, this value should be set to a large number (internally it is limited to 40, the maximum cell hierarchy depth) which will ensure that all via-induced connections are found. This can dramatically increase extraction time.

The value of this variable tracks the **Via search depth** entry area in the **Net Config** page of the **Extraction Setup** panel from the **Extract Menu**.

ViaConvex
Value: boolean.
This applies when checking for connectivity through a via during extraction. When set, all non-rectangular vias are assumed to be convex polygons. The test region is taken as a small rectangle centered on the via bounding box. This simplifies and should speed testing. It is intended specifically for circular vias, as used in superconductive electronics. It has no effect on rectangular vias. It should **not** be set if any vias are non-convex polygons, as incorrect results may occur.

Whether or not this variable is set tracks the state of the **Assume convex vias** check box in the **Net Config** page of the **Extraction Setup** panel from the **Extract Menu**.

E.26 Extraction Menu Commands

The **!set** variables below affect the commands found in the **Extract Menu**.

QpathGroundPlane

Value: integer 0–2.

This variable controls how the **"Quick" Path** command in the extraction **Path Selection Control** panel uses the inverted ground plane. Normally, during extraction, if the GroundPlaneClear keyword has been given, an inverted ground plane is created on a temporary layer for internal use. Since the **"Quick" Path** mode operates outside of the extraction system, the inverted ground plane may or may not be available. The choices are:

0

Use the inverted ground plane if available. This is the default. If an inverted ground plane has already been created and is current, it will be used when determining paths. If the ground plane does not have a current inversion, the absence of the layer will imply a ground contact, as in extraction without the MultiNet keyword. This choice avoids the sometimes lengthly inversion computation, but makes use of the inversion if it has already been done.

1

Create the inverted ground plane if necessary, and use it. If the extraction system would use an inverted ground plane, it will be created if not already present and current. The path selection will include the inverted layer.

2

The **"Quick" Path** mode will never use the inverted ground plane.

This variable tracks the state of the **"Quick" Path ground plane handling** menu in the **Path Selection Control** panel.

QpathUseConductor

Value: boolean.

By default, when this variable is not set, only objects on layers with the Routing attribute applied will be considered for inclusion in the path extracted with the **"Quick" Path** button in the **Path Selection Control** panel, which is obtained from the **Net Selections** button in the **Extract Menu**. If this variable is set, objects on layers with the Conductor attribute will be allowed. The Routing attribute implies Conductor, but may be more restrictive.

This variable tracks the state of the **"Quick" Path use Conductor** check box in the **Path Selection Control** panel.

EnetNet

Value: boolean.

If set, the netlist in internal format is incuded when writing output in the **Dump Elec Netlist** command. This variable corresponds to the net check box available in that command.

EnetSpice

Value: boolean.

If set, SPICE output is included in the file produced from the **Dump Elec Netlist** command. This variable corresponds to the spice check box available in that command.

EnetBottomUp

Value: boolean.

When set, the electrical netlist file (produced by the **Dump Elec Netlist** command) order will be leaf-to-root, i.e., subcells will be listed first. If not set, the reverse order is used.

PnetNet

Value: boolean.

If set, the extracted netlist listing in the internal format is included in output from the **Dump Phys Netlist** command. This variable corresponds to the net check box available in that command.

PnetDevs

Value: boolean.

If set, the extracted device listing in internal format is included in output from the **Dump Phys Netlist** command. This variable corresponds to the devs check box available in that command.

PnetSpice

Value: boolean.

If set, the SPICE listing of extracted devices is included in output from the **Dump Phys Netlist** command. This variable corresponds to the spice check box available in that command.

PnetBottomUp

Value: boolean.

When set, the physical netlist file (produced by the **Dump Phys Netlist** command) order will be leaf-to-root, i.e., subcells will be listed first. If not set, the reverse order is used.

PnetShowGeometry

Value: boolean.

If set, the net field (if activated) in the file produced from the **Dump Phys Netlist** command will include a listing of the objects that comprise the wire net. The listing is in modified CIF syntax where 1000 units per micron is used. This variable corresponds to the show geometry check box available in that command.

PnetIncludeWireCap

Value: boolean.

If set, the spice field (if activated) in the file produced from the **Dump Phys Netlist** command will include capacitors representing the computed wire net capacitance to ground. The Routing layers must have the Capacitance keyword applied in the technology file. The added capacitors have a special prefix "`C@NET`" which allows them to be subsequently recognized as wire net capacitors by *Xic*. This variable corresponds to the include wire cap check box available in that command.

PnetListAll

Value: boolean.

In files produced with the **Dump Phys Netlist** command, references to subcells that are flattened or wire-only are normally not listed. If this variable is set, these cells are included in the listing, which may be useful for debugging. This variable corresponds to the include all devs check box available in that command.

PnetNoLabels

Value: boolean.

When set, output from the **Dump Phys Netlist** command will use group numbers to designate non-global nets. When not set, output will use group names as provided by net name labels (see 16.5) where found. This variable mirrors the state of the **ignore labels** check box in the **Dump Phys Netlist** panel.

PnetVerbose

Value: boolean.

This boolean variable is intended to enable additional information when printing output from the **Dump Phys Netlist** command. Presently, it only applies when printing the device table (PnetDevs is set). It will print additional information about multi-component (merged) devices. This variable mirrors the state of the **devs verbose** check box in the **Dump Phys Netlist** panel, available from the button in the **Extract Menu**.

SourceAllDevs

Value: boolean.

In the **Source SPICE** command, ordinarily only devices which have fixed (user-specified) device names will have properties updated. This is to avoid errors, since the internally generated names can change, and may not match those in the SPICE file. If this variable is set, the default action is to update all devices. This variable corresponds to the all devs check box available in that command.

SourceCreate
Value: boolean.
In the **Source SPICE** command, if this variable is set, the default action is to create missing devices. Otherwise, device parameters may be updated, but no new devices are created. This variable corresponds to the create check box available in that command.

SourceClear
Value: boolean.
In the **Source SPICE** command, if this variable is set the default action is to discard the existing contents of the electrical part of the cell before updating. This variable corresponds to the clear check box available in that command.

SourceGndDevName
Value: string.
This variable specifies the name of the ground terminal device to use when devices are created and placed in the **Source SPICE** and (consequently) the **Source Physical** extraction commands. If not set, the name "`gnd`" will be assumed. If this variable is set to a name, a ground device of that name must appear in the device library file.

SourceTermDevName
Value: string.
This variable specifies the name of the terminal device to use when devices are created and placed in the **Source SPICE** and (consequently) the **Source Physical** extraction commands. If not set, the name "`tbar`" will be assumed, if that name is found for a terminal device in the device library. If not found, the name "`vcc`" will be assumed. If this variable is set to a name, that name must match the name of a terminal device in the device library file.

NoExsetAllDevs
Value: boolean.
In the **Source Physical** command, if this variable is set, only devices that have a permanent (user-supplied) name will be updated. If not set, all devices will be updated. This variable corresponds to the all devs check box available in that command, with inverse logic.

NoExsetCreate
Value: boolean.
The default behavior of the **Source Physical** command is to create missing devices. Setting this variable will change the default action to no device creation. This variable corresponds to the create check box available in that command, with inverse logic.

ExsetClear
Value: boolean.
When set, the electrical cells are cleared before updating with the **Source Physical** command. This implies create, i.e., new devices will be created since the cell is empty. This variable corresponds to the clear check box available in that command.

ExsetIncludeWireCap
Value: boolean.
When set, computed routing capacitors will be updated or created in the electrical database when

using the **Source Physical** command. These capacitors have a name prefix of "C@NET". This variable corresponds to the include wire cap check box available in that command.

ExsetNoLabels

Value: boolean.

When set, output from the **Source Physical** command will use group numbers to designate non-global nets. When not set, output will use group names as provided by net name labels (see 16.5) where found.

LvsFailNoConnect

Value: boolean.

During LVS analysis, the electrical (schematic) part of the design is used as the basis for recursion through the hierarchy. Thus, physical subcells that have no connection to the circuit will not be detected, and are basically ignored. However, an explicit test is performed for such cells, and those found will be listed in the LVS report. If this variable is set, the presence of such cells will force LVS failure, otherwise they are ignored for comparison purposes.

This variable tracks the state of the **fail if unconnected physical subcells** check box in the panel brought up by the **Dump LVS** button in the **Extract Menu**.

PathFileVias

Value: boolean or string.

This variable determines whether and how vias are included in the files produced with the **Save path to file** button in the **Path Selection Control** panel from the **Net Selections** button in the **Extract Menu**. It tracks (and sets) the state of the **Path file contains vias** and **Path file contains check layers** check boxes in the panel.

If not set, via layers will not be included in the file, only the conductors will appear. If set as a boolean (i.e., to no value), the via layers will be included, but not the check layers. If set to any text, the check layers will also be included.

E.27 Capacitance Extraction Interface

The following variables apply to the capacitance extraction interface described in 16.17.1. Most of these are associated with entry fields in the **Cap Extraction** panel (see16.17.2), which is brought up with the **Extract C** button in the **Extract Menu**.

FcArgs

Value: string.

This variable can be set to a string, which will be included in the argument list when capacitance extraction is initiated through the interface, with the **Run Extraction** button in the **Run** page of the **Cap Extraction** panel, or through the **!fc** command. The variable tracks the **FcArgs** text entry area in the **Run** page of the **Cap Extraction** panel, from where the variable is most conveniently set or edited.

If the interface detects that *FasterCap* from FastFieldSolvers.com is being used, and this entry is empty, the default argument string

```
-b -a0.01
```

will be imposed. A "-b" option will always be added if missing from the *FasterCap* arguments list, as this argument is necessary for correct *FasterFap* operation in this mode. The "-a" option is almost always used, as it specifies auto-refinement, however it is technically not necessary and won't be imposed if not given, except in the case where no arguments are given at all.

FcForeg

Value: boolean.

If this variable is set, then the **Run Extraction** button in the **Cap Extraction** panel **Run** page will initiate a process running in the foreground. If not set, jobs are run in the background, so that the user can continue using *Xic* while the run is in progress.

It is not clear why there would be any reason to run in the foreground, except possibly for debugging.

This variable controls, and is controlled by, the setting of the **Run in foreground** check box in the **Run** page of the **Cap Extraction** panel from the **Extract Menu**.

FcLayerName

Value: string.

The capacitance extraction interface uses a special layer for masking of objects to be included in the capacitance extraction run. By default, this layer is named "`FCAP`". If any shapes exist on this layer in the current cell hierarchy, all objects will be clipped by these shapes before capacitance extraction. If no shapes are found on this layer, then all objects in the current cell hierarchy will be included in capacitance extraction.

If this variable is set to the name of an existing layer name in the layer table, that layer will do the clipping.

FcMonitor

Value: boolean.

If this variable is set, then the standard output from the running capacitance extraction program is printed in the console, in addition to being saved in a file. The console is the shell window from which *Xic* was started. This allows the user to monitor the run, and abort if something isn't correct.

This will also apply if the program is being run in the foreground, however operation is a bit different. In this case, a "`| tee`" is added to the command string ahead of the output file name. There are two implications: the text will be block buffered, and therefor won't appear in the window immediately, and in Windows, there is no native `tee` command so that the operation may fail. However, a `tee` command is provided with the Cygwin tools, and there are other sources. In the normal case of running in the background, output will again be block buffered under Windows, but there is no requirement for a `tee` command.

This variable mirrors the state of the **Out to console** check box in the **Run** page of the **Cap Extraction** panel from the **Extract Menu**.

FcPanelTarget

Value: real number 1e3 – 1e6.

When **not** using a capacitance extraction program that provides automatic refinement, such as *FasterCap* from `FastFieldSolvers.com`, this provides a crude panel refinement capability. This variable provides a number, and the interface will attempt to split all panels into equal area pieces, where the total number of pieces is the number given. The refined panels are output into the list file, which consequently can grow large.

When not set, no such refinement is done. It should not be set for normal use of *FasterCap*, but is needed if using the Whiteley Research version of *FastCap* or similar.

FcPath

Value: directory path string.

This variable can be set to a full path to the capacitance extraction program executable.

If this is not set, *Xic* will attempt to use "`/usr/local/bin/fastcap`" as the *FastCap* program (or "`/usr/local/bin/fastcap.exe`" in Windows). If this executable does not exist, *Xic* will attempt to find "`fastcap`" (or "`fastcap.exe`" in Windows) in the shell search path when running in the foreground, and background runs will fail.

This tracks the setting of the text entry field in the **Run** page of the **Cap Extraction** panel.

FcPlaneBloat

Value: real number 0.0 – 100.0.
If set to a positive value, the substrate is modeled to extend horizontally outward by this value beyond the bounding box of the extracted geometry. See the discussion in the interface description in 16.17.1 for more information. If not set, no dimensional change is assumed.

FcUnits

Value: units string.
This variable can be used to specify the length units used in generated capacitance extraction input files. The variable can be set to a string consisting of one or the abbreviations "`m`" (meters), "`cm`" (centimeters), "`mm`" (millimeters), "`um`"" (microns), "`in`" (inches), and "`mils`". The long form word will also be accepted. This variable is most conveniently manipulated with the choice menu found in the **Cap Extraction** panel **Params** page.

E.28 Inductance/Resistance Extraction Interface

The following variables apply to the inductance/resistance extraction (*FastHenry* interface). Most of these are associated with entry fields in the **LR Extraction** panel, which is brought up with the **Extract LR** button in the **Extract Menu**.

FhArgs

Value: string.
This value can be set to a string, which will be included in the argument list when *FastHenry* is initiated with the **Run FastHenry** button in the **LR Extraction** panel **Run** page. The variable is most conveniently manipulated with the text entry field in the **LR Extraction** panel **Run** page.

FhDefaults

Value: string.
If set to a string, the value will be used in a `.DEFAULT` line in the *FastHenry* input file created by the interface. The variable is most conveniently manipulated with the text entry field in the **LR Extraction** panel **Run** page. See the *FastHenry* documentation describing the syntax and options for the applicable text.

FhDefNhinc

Value: integer 1 – 20.
Provide a default value for the `nhinc` parameter as used by *FastHenry*. This is overridden by values specified with the `FH_nhinc` technology keyword for layers, unless the FhOverride variable is set, in which case the this variable has precedence. This tracks the **FhDefNhinc** entry in the **Params** page of the **LR Extraction** panel.

FhDefRh

Value: real 0.5 – 4.0.
Provide a default value for the `rh` parameter as used by *FastHenry*. This is overridden by values specified with the `FH_rh` technology keyword for layers, unless the FhOverride variable is set, in which case the this variable has precedence. This tracks the **FhDefRh** entry in the **Params** page of the **LR Extraction** panel.

FhForeg

Value: boolean.

If this variable is set, then the **Run FastHenry** button in the **LR Extraction** panel **Run** page will initiate a *FastHenry* run in the foreground. If not set, jobs are run in the background, so that the user can continue using *Xic* while the run is in progress.

It is not clear why there would be any reason to run in the foreground, except possibly for debugging.

This variable controls, and is controlled by, the setting of the **Run in foreground** check box in the **Run** page of the **LR Extraction** panel from the **Extract Menu**.

FhFreq

Value: string.

This variable can be used to specify the evaluation frequencies used for *FastHenry*, as included in a generated input file, or when initiating a run. The format is the same as is used in the *FastHenry* input format:

`fmin=`*start_freq* `fmax=`*stop_freq* [`ndec=`*num*]

The frequencies are floating point numbers given in hertz, and the `ndec` parameter, if given, specifies the number of intermediate frequencies to evaluate. If the third field is not set, evaluation is at the start and stop frequencies only, or at the single frequency if both are the same. If the variable is not set, the evaluation is at a single frequency of one kilohertz. This variable is most conveniently manipulated with the text entry fields in the **LR Extraction** panel **Run** page.

FhLayerName

Value: string.

The inductance/resistance extraction interface uses a special layer for masking of objects to be included in the extraction run. By default, this layer is named "`FHRY`". If any shapes exist on this layer in the current cell hierarchy, all objects will be clipped by these shapes before inductance/resistance extraction. If no shapes are found on this layer, then all objects in the current cell hierarchy will be included in extraction.

If this variable is set to the name of an existing layer name in the layer table, that layer will do the clipping.

FhManhGridCnt

Value: real number 1e2–1e5.

When a non-Manhattan polygon is "Manhattanized" for *FastHenry*, it is converted to an approximating Manhattan polygon. This variable can be used to set the minimum rectangle width and height used in the decomposition. This value is given by

`sqrt(`*area_of_interest*`/FhManhGridCnt)`

If not set, a value of 1000 is used. Larger values are more accurate but slow processing, sometimes dramatically. The *area_of_interest* is the layout area being processed for input to *FastHenry*.

This variable is most conveniently manipulated with the text input field in the **LR Extraction** panel **Params** page.

FhMonitor

Value: boolean.

If the variable is set, then the standard output from the running *FastHenry* program is printed in the console, in addition to being saved in a file. The console is the shell window from which *Xic* was started. This allows the user to monitor the run, and abort if something isn't correct.

This will also apply if the program is being run in the foreground, however operation is a bit different. In this case, a "| tee" is added to the command string ahead of the output file name. There are two implications: the text will be block buffered, and therefor won't appear in the window immediately, and in Windows, there is no native tee command so that the operation may fail. However, a tee command is provided with the Cygwin tools, and there are other sources. In the normal case of running in the background, output will again be block buffered under Windows, but there is no requirement for a tee command.

This variable mirrors the state of the **Out to console** check box in the **Run** page of the **LR Extraction** panel from the **Extract Menu**.

FhOverride

Value: boolean.
When set, values of the FhDefNhinc and FhDefRh variables will override values provided by FH_nhinc and FH_rh technology layer parameters to use for nhinc and rh parameters in *FastHenry* input. This tracks the state of the **Override Layer NHINC, RH** button in the **Params** page of the **LR Extraction** panel.

FhPath

Value: directory path string.
This variable can be set to a full path to the *FastHenry* executable.

If this is not set, *Xic* will attempt to use "/usr/local/bin/fasthenry" as the *FastHenry* program (or "/usr/local/bin/fasthenry.exe" in Windows). If this executable does not exist, *Xic* will attempt to find "fasthenry" (or "fasthenry.exe" in Windows) in the shell search path when running in the foreground, and background runs will fail.

This tracks the setting of the text entry field in the **Run** page of the **LR Extraction** panel.

FhUnits

Value: units string.
This variable can be used to specify the length units used in generated *FastHenry* input files. The variable can be set to a string consisting of one or the abbreviations "m" (meters), "cm" (centimeters), "mm" (millimeters), "um" (microns), "in" (inches), and "mils". The long form word will also be accepted. This variable is most conveniently manipulated with the choice menu found in the **LR Extraction** panel **Params** page.

FhUseFillament

Value: units boolean.
If set, *FastHenry* will decompose segments into filaments according to the given nhinc and rh parameters. If not set, the interface will automatically slice segments according to the same parameters, without further refinement by *FastHenry*. This tracks the state of the **Use FastHenry Internal NHINC, RH** button in the **Params** page of the **LR Extraction** panel.

FhVolElMin

Value: units real 0 – 1.0, default 0.1.
The minimum rectangle edge length is this factor multiplying the maximum edge length. This tracks the **FhVolElMin** entry in the **Params** page of the **LR Extraction** panel.

FhVolElTarget

Value: real number 1e2 – 1e5, default 1e3.
This controls refinement for *FastHenry*. The total volume of all conductors is divided by this value, and the cube root taken to provide a length. Volume elements are split so that no edge is longer than this length. The total number of volume elements is approximately the value of this variable. Each volume element contains six segments, connecting the center node to each face node.

FhVolEnable

Value: boolean.

Enable segment refinement. This tracks the state of the **Enable** button in the **Params** page of the **LR Extraction** panel.

E.29 Help System

The following **!set** variables affect the help system.

HelpDefaultTopic

Value: string.

If this variable is set to an empty string (i.e., as a boolean) the default help window which normally appears when the **Help** button in the **Help Menu** is pressed does not appear. The help mode is still set, so help can be obtained in the usual way by pressing buttons or through other actions, only the initial window is suppressed.

Otherwise, this variable can be set to a URL or help system keyword, which will be shown in the initial window when the **Help** menu button is pressed.

HelpMultiWin

Value: boolean.

This variable, when set, causes the help system to use a new window for each menu item or screen element clicked on in help mode. If not set, the original help window is reused.

The state of this variable tracks the **Multi-Window Mode** button in the **Help Menu**.

See also the HelpPath and DocsDir variables in E.3.

Appendix F

Interface Functions

There is a growing library of user interface functions which control various aspects of *Xic* for use in scripts.

Functions that manipulate objects in the database use a coordinate system based in microns (1 micron usually equals 1000 database units). All coordinates are real values.

There are two levels of run-time error reporting. For serious errors, a message is emitted to the controlling terminal, and the script terminates. Most interface functions will generate this type of error only in response to bad arguments, meaning usually arguments of the wrong type. Less serious errors simply cause the function to return, returning a value that indicates that the operation was unsuccessful. Many of the functions return 1 if successful, or 0 if not successful. In some cases where a string is normally returned, a null string return indicates an error occurred. It is up to the user to test the return values for success or failure.

When the documentation specifies that a null string value is acceptable as a function argument, the value zero can be passed instead of a string variable. The token `NULL`, which is predefined as 0, can be used equivalently.

The tables below list the collections of interface functions presently available, by category and sub-category. Most of these functions return a value. In the descriptions, if a value is returned, the type, in parentheses, is indicated ahead of the function name.

The first group of main module functions:

Main Functions 1	
Current Cell	
`Edit(`*name*`, `*symname*`)`	Edit cell
`OpenCell(`*name*`, `*symname*`, `*curcell*`)`	Read file into memory
`TouchCell(`*cellname*`, `*curcell*`)`	Create cell in memory
`RegisterSubMasters(`*archive*`)`	Pre-load pcell sub-masters to resolve as archive is read
`Push(`*object_handle*`)`	Make a subcell the current cell
`PushElement(`*object_handle*`, `*xind*`, `*yind*`)`	Make an arrayed subcell element the current cell
`Pop()`	Make parent cell the current cell
`NewCellName()`	Return empty new cell name
`CurCellName()`	Return current cell name
`TopCellName()`	Return cell name at top of editing hierarchy

`FileName()`	Return file name for current cell
`CurCellBB(`*array*`)`	Return current cell bounding box
`SetCellFlag(`*cellname*`, `*flagname*`, `*set*`)`	Set the state of a cell flag
`GetCellFlag(`*cellname*`, `*flagname*`)`	Get cell flag state
`Save(`*newname*`)`	Save to disk
`UpdateNative(`*dir*`)`	Save modified hierarchy cells as native
Cell Info	
`CellBB(`*cellname*`, `*array*` [, `*symbolic*`])`	Obtain cell bounding box
`ListSubcells(`*cellname*`, `*depth*`, `*array*`, `*incl_top*`)`	List subcells in area to depth
`ListParents(`*cellname*`)`	List instantiating cells
`InitGen()`	Return handle to subcell name list
`CellsHandle(`*cellname*`, `*depth*`)`	Return handle to subcell name list
`GenCells(`*handle*`)`	Return name from name list
Database	
`Clear(`*cellname*`)`	Delete cells from memory
`ClearAll(`*clear_tech*`)`	Delete all cells and reinitialize
`IsCellInMem(`*cellname*`)`	Check if cell is in memory
`IsFileInMem(`*filename*`)`	Check if cell from file is in memory
`NumCellsInMem()`	Count cells in memory
`ListCellsInMem(`*options_str*`)`	List names of cells in memory
`ListTopCellsInMem()`	List names of top-level cells in memory
`ListModCellsInMem()`	List names of modified cells in memory
`ListTopFilesInMem()`	List source files of top-level cells in memory
Symbol Tables	
`SetSymbolTable(`*tabname*`)`	Switch to new or existing symbol table
`ClearSymbolTable(`*destroy*`)`	Clear or destroy current symbol table
`CurSymbolTable()`	Return the name of the current symbol table
Display	
`Window(`*x*`, `*y*`, `*width*`, `*win*`)`	Set display window view
`GetWindow()`	Return window containing pointer
`GetWindowView(`*win*`, `*array*`)`	Return window view area coordinates
`GetWindowMode(`*win*`)`	Return window display mode
`Expand(`*win*`, `*string*`)`	Set expansion status
`Display(`*display_string*`, `*win_id*`, `*l*`, `*b*`, `*r*`, `*t*`)`	Exportable rendering service
`FreezeDisplay(`*freeze*`)`	Turn off/on graphics screen updates
`Redraw(`*win*`)`	Redraw the window
Exit	
`Exit()`	Exit script
`Halt()`	Exit script
Annotation	
`AddMark(`*type*`, `*arguments*` ...)`	Show a user-specified mark
`EraseMark(`*id*`)`	Erase a mark
`DumpMarks(`*filename*`)`	Dump current cell marks to file
`ReadMarks(`*filename*`)`	Read marks from file
Ghost Rendering	
`PushGhost(`*array*`, `*numpts*`)`	Register ghost-drawn polygon
`PushGhostBox(`*left*`, `*bottom*`, `*right*`, `*top*`)`	Register ghost-drawn box

`PushGhostH(`*object_handle*`,` *all*`)`	Register ghost-drawn outlines
`PopGhost()`	Unregister ghost-drawn figure
`ShowGhost(`*type*`)`	Show ghost-drawn figures
Graphics	
`GRopen(`*display*`,` *window*`)`	Open a graphics context
`GRcheckError()`	Return graphics error status
`GRcreatePixmap(`*handle*`,` *width*`,` *height*`)`	Return a new pixmap id
`GRdestroyPixmap(`*handle*`,` *pixmap*`)`	Free pixmap
`GRcopyDrawable(`*handle*`,` *dst*`,` *src*`,` *xs*`,` *ys*`,` *ws*`,` *hs*`,` *x*`,` *y*`)`	Copy area between drawables
`GRdraw(`*handle*`,` *l*`,` *b*`,` *r*`,` *t*`)`	Render cell
`GRgetDrawableSize(`*handle*`,` *drawable*`,` *array*`)`	Return size of drawable
`GRresetDrawable(`*handle*`,` *drawable*`)`	Switch drawable in context
`GRclear(`*handle*`)`	Clear window
`GRpixel(`*handle*`,` *x*`,` *y*`)`	Draw pixel
`GRpixels(`*handle*`,` *array*`,` *num*`)`	Draw pixels
`GRline(`*handle*`,` *x1*`,` *y1*`,` *x2*`,` *y2*`)`	Draw line
`GRpolyLine(`*handle*`,` *array*`,` *num*`)`	Draw path
`GRlines(`*handle*`,` *array*`,` *num*`)`	Draw lines
`GRbox(`*handle*`,` *l*`,` *b*`,` *r*`,` *t*`)`	Draw box
`GRboxes(`*handle*`,` *array*`,` *num*`)`	Draw boxes
`GRarc(`*handle*`,` *x0*`,` *y0*`,` *rx*`,` *ry*`,` *theta1*`,` *theta2*`)`	Draw arc
`GRpolygon(`*handle*`,` *array*`,` *num*`)`	Draw polygon
`GRtext(`*handle*`,` *text*`,` *x*`,` *y*`,` *flags*`)`	Draw text
`GRtextExtent(`*handle*`,` *text*`,` *array*`)`	Return text size
`GRdefineColor(`*handle*`,` *red*`,` *green*`,` *blue*`)`	Return color code
`GRsetBackground(`*handle*`,` *pixel*`)`	Set default background color
`GRsetWindowBackground(`*handle*`,` *pixel*`)`	Set window background color
`GRsetColor(`*handle*`,` *pixel*`)`	Set foreground color
`GRdefineLinestyle(`*handle*`,` *index*`,` *mask*`)`	Define a line style
`GRsetLinestyle(`*handle*`,` *index*`)`	Set current line style
`GRdefineFillpattern(`*handle*`,` *index*`,` *nx*`,` *ny*`,` *array_string*`)`	Define a fill pattern
`GRsetFillpattern(`*handle*`,` *index*`)`	Set current fill pattern
`GRupdate(`*handle*`)`	Update rendering
`GRsetMode(`*handle*`,` *mode*`)`	Set drawing mode
Hard Copy	
`HClistDrivers()`	Return list of available drivers
`HCsetDriver(`*driver*`)`	Set current driver
`HCgetDriver()`	Return current driver name
`HCsetResol(`*resol*`)`	Set current driver resolution
`HCgetResol()`	Return current driver resolution
`HCgetResols(`*array*`)`	Return available driver resolutions
`HCsetBestFit(`*best_fit*`)`	Set “best fit” mode
`HCgetBestFit()`	Return “best fit” mode
`HCsetLegend(`*legend*`)`	Set “legend” mode
`HCgetLegend()`	Return “legend” mode

HCsetLandscape(*landscape*)	Set “landscape” mode
HCgetLandscape()	Return “landscape” mode
HCsetMetric(*metric*)	Set “metric” mode
HCgetMetric()	Return “metric” mode
HCsetSize(*x*, *y*, *w*, *h*)	Set rendering area
HCgetSize(*array*)	Return rendering area
HCshowAxes(*style*)	Set axes display style
HCshowGrid(*show*, *mode*)	Set grid displayed or not
HCsetGridInterval(*spacing*, *mode*)	Set grid spacing
HCsetGridStyle(*linemod*, *mode*)	Set grid line style
HCsetGridCrossSize(*xsize*, *mode*)	Set grid “dot” cross size
HCsetGridOnTop(*on_top*, *mode*)	Draw grid above or below geometry
HCdump(*l*, *b*, *r*, *t*, *filename*, *command*)	Generate output
HCerrorString()	Retrun error message
HClistPrinters()	List MS Windows printers
HCmedia(*index*)	Set MS Windows page size
Keyboard	
ReadMapfile(*mapfile*)	Read a keyboard mapping file
Libraries	
OpenLibrary(*path_name*)	Open a library file
CloseLibrary(*path_name*)	Close an open library
OpenAccess	
OaVersion()	Get OpenAccess version string
OaIsLibrary(*libname*)	Check if argument is a library
OaListLibraries()	Return list of libraries
OaListLibCells(*libname*)	Return list of cells in library
OaListCellViews(*libname*, *cellname*)	Return list of views in cell
OaIsLibOpen(*libname*)	Check if library is open
OaOpenLibrary(*libname*)	Open an OpenAccess library
OaCloseLibrary(*libname*)	Close an open OpenAccess library
OaIsOaCell(*libname*, *open_only*)	Check if cell can be resolved
OaIsCellInLib(*libname*, *cellname*)	Check if cell exists in library
OaIsCellView(*cellname*, *viewname*, *open_only*)	Check if view exists in cell
OaIsCellViewInLib(*libname*, *cellname*, *viewname*)	Check if view of cell exists in cell
OaCreateLibrary(*libname*, *techlibname*)	Create new library
OaBrandLibrary(*libname*, *branded*)	Set or unset writability from *Xic*
OaIsLibBranded(*libname*)	Check if library writable from *Xic*
OaDestroy(*libname*, *cellname*, *viewname*)	Destroy library, cell, or view
OaLoad(*libname*, *cellname*)	Load cell into *Xic*
OaReset()	Clear table of cells already loaded
OaSave(*libname*, *allhier*)	Save current cell to OpenAccess
OaAttachTech(*libname*, *techlibname*)	Attach the technology from another library
OaGetAttachedTech(*libname*)	Return the name of attached library
OaHasLocalTech(*libname*)	Check if library has local tech database
OaCreateLocalTech(*libname*)	Create a local tech database in library
OaDestroyTech(*libname*, *unattach_only*)	Destroy/remove technology object

Mode	
`Mode(`*window*`, `*mode*`)`	Set physical or electrical mode
`CurMode(`*window*`)`	Return current mode
Prompt Line	
`StuffText(`*string*`)`	Register text for future access
`TextCmd(`*string*`)`	Execute a prompt line command
`GetLastPrompt()`	Return most recent prompt line message
Scripts	
`ListFunctions()`	Return list of library file functions
`Exec(`*script*`)`	Execute a script
`SetKey(`*password*`)`	Set the current password for script decryption
`HasPython()`	Return true if Python is available
`RunPython(`*command*`)`	Run a Python script
`RunPythonModFunc(`*module*`, `*function* `[, `*arg* `...])`	Execute a Python module function
`ResetPython()`	Reset the Python interpreter
`HasTcl()`	Return true if Tcl is available
`HasTk()`	Return true if Tcl and Tk are available
`RunTcl(`*command* `[, `*arg* `...])`	Run a Tcl/Tk script
`ResetTcl()`	Reset the Tcl/Tk interpreter
`HasGlobalVariable(`*globvar*`)`	Test if global variable
`GetGlobalVariable(`*globvar*`)`	Return value of global variable
`GetGlobalVariable(`*globvar*`, `*value*`)`	Set value of global variable
Technology File	
`GetTechName()`	Return technology name
`GetTechExt()`	Return technology file extension
`SetTechExt(`*extension*`)`	Define effective technology file extension
`TechParseLine(`*line*`)`	Parse text in technology file format
`TechGetFkeyString(`*fkeynum*`)`	Return function key encoding string
`TechSetFkeyString(`*fkeynum*`, `*string*`)`	Set function key encoding
Variables	
`Set(`*name*`, `*string*`)`	Set a variable
`Unset(`*name*`)`	Unset a variable
`PushSet(`*name*`, `*string*`)`	Set a variable, allow revert
`PopSet(`*name*`)`	Revert `PushSet`
`SetExpand(`*string*`, `*use_env*`)`	Perform variable substitution
`Get(`*name*`)`	Return variable contents
`JoinLimits(`*flags*`)`	Set or remove join operation limits
***Xic* Version**	
`VersionString()`	Return current *Xic* version

The second group of main module functions:

Main Functions 2	
Arrays	
`ArrayDims(`*out_array*`, `*array*`)`	Get array dimensions
`ArrayDimension(`*out_array*`, `*array*`)`	Get array dimensions
`GetDims(`*array*`, `*out_array*`)`	Get array dimensions

`DupArray(`*dest_array*`,` *src_array*`)`	Copy an array
`SortArray(`*array*`,` *size*`,` *descend*`,` *indices*`)`	Sort array elements
Bitwise Logic	
`ShiftBits(`*bits*`,` *val*`)`	Shift bit field
`AndBits(`*bits1*`,` *bits2*`)`	AND operation
`OrBits(`*bits1*`,` *bits2*`)`	OR operation
`XorBits(`*bits1*`,` *bits2*`)`	XOR operation
`NotBits(`*bits*`)`	NOT operation
Error Reporting	
`GetError()`	Return error message
`AddError(`*string*`)`	Save error string
`GetLogNumber()`	Return current message index
`GetLogMessage(`*message_num*`)`	Return string for message index
`AddLogMessage(`*string*`,` *error*`)`	Add message to log
Generic Handle Functions	
`NumHandles()`	Returns the number of active handles
`HandleContent(`*handle*`)`	Returns count of list items
`HandleTruncate(`*handle*`,` *count*`)`	Truncate a list of items
`HandleNext(`*handle*`)`	Advance list to next item
`HandleDup(`*handle*`)`	Duplicate a handle and list
`HandleDupNitems(`*handle*`,` *count*`)`	Duplicate a handle and list, truncating list
`H(`*scalar*`)`	Create temporary handle from scalar
`HandleArray(`*handle*`,` *array*`)`	Write an array of handles to list elements
`HandleCat(`*handle1*`,` *handle2*`)`	Add *handle2* list to end of *handle1* list
`HandleReverse(`*handle*`)`	Reverse list order
`HandlePurgeList(`*handle1*`,` *handle2*`)`	Remove from second list items in first
`Close(`*handle*`)`	Close a handle
`CloseArray(`*array*`,` *size*`)`	Close an array of handles
Memory Management	
`FreeArray(`*array*`)`	Free memory used by array
`CoreSize()`	Return kilobytes used by program
Script Variables	
`Defined(`*variable*`)`	Check if variable is defined
`TypeOf(`*variable*`)`	Return variable type
Path Manipulation and Query	
`PathToEnd(`*path_name*`,` *dir*`)`	Modify search path
`PathToFront(`*path_name*`,` *dir*`)`	Modify search path
`InPath(`*path_name*`,` *dir*`)`	Check if directory is in search path
`RemovePath(`*path_name*`,` *dir*`)`	Remove directory from the search path
Regular Expressions	
`RegCompile(`*regex*`,` *case_insens*`)`	Compile regular expression
`RegCompare(`*regex_handle*`,` *string*`,` *array*`)`	Regular expression evaluation
`RegError(`*regex_handle*`)`	Return error string
String List Handles	
`StringHandle(`*string*`,` *sepchars*`)`	Return handle to string tokens
`ListHandle(`*arglist*`)`	Return handle to string arguments
`ListContent(`*stringlist_handle*`)`	Return referenced string
`ListReverse(`*stringlist_handle*`)`	Reverse order of strings in list

`ListNext(`*stringlist_handle*`)`	Return referenced string and advance to next
`ListAddFront(`*stringlist_handle*`, `*string*`)`	Add string to list
`ListAddBack(`*stringlist_handle*`, `*string*`)`	Add string to list
`ListAlphaSort(`*stringlist_handle*`)`	Sort string list
`ListUnique(`*stringlist_handle*`)`	Remove duplicates from list
`ListFormatCols(`*stringlist_handle*`, `*columns*`)`	Format strings into columns
`ListConcat(`*stringlist_handle*`, `*sepchars*`)`	Create single string from list
`ListIncluded(`*stringlist_handle*`, `*string*`)`	Check if string is in list
String Manipulation and Conversion	
`Strcat(`*string1*`, `*string2*`)`	String concatenation
`Strcmp(`*string1*`, `*string2*`)`	String comparison
`Strncmp(`*string1*`, `*string2*`, `*n*`)`	String comparison, fixed length
`Strcasecmp(`*string1*`, `*string2*`)`	String comparison, case insensitive
`Strncasecmp(`*string1*`, `*string2*`, `*n*`)`	String comparison, case insensitive, fixed length
`Strdup(`*string*`)`	String copy
`Strtok(`*str*`, `*sep*`)`	String tokenization
`Strchr(`*string*`, `*char*`)`	Return pointer to first instance of character
`Strrchr(`*string*`, `*char*`)`	Return pointer to last instance of character
`Strstr(`*string*`, `*substring*`)`	Return pointer to first instance of substring
`Strpath(`*string*`)`	Return pointer to filename in path
`Strlen(`*string*`)`	Return length of string
`Sizeof(`*arg*`)`	Return string length or array size
`ToReal(`*string*`)`	Convert string to number
`ToString(`*real*`)`	Convert number to string
`ToStringA(`*real*`, `*digits*`)`	Convert number to string using SPICE notation
`ToFormat(`*format*`, `*arg_list*`)`	Print variables according to format string
`ToChar(`*integer*`)`	Convert character constant to string representation
Current Directory	
`Cwd(`*path*`)`	Set current directory
`Pwd()`	Return current directory
Date and Time	
`DateString()`	Return the date/time
`Time()`	Return system-encoded time
`MakeTime(`*array*`, `*gmt*`)`	Create system-encoded time from values
`TimeToString(`*time*`, `*gmt*`)`	Return string from system-encoded time
`TimeToVals(`*time*`, `*gmt*`, `*array*`)`	Parse system-encoded time
`MilliSec()`	Return elapsed time in milliseconds
`StartTiming(`*array*`)`	Initialize resource timing
`StopTiming(`*array*`)`	Obtain resource times
File System Interface	
`Glob(`*pattern*`)`	Perform global expansion
`Open(`*file*`, `*mode*`)`	Open a file for read/write
`Popen(`*command*`, `*mode*`)`	Open a process for read/write
`Sopen(`*host*`, `*port*`)`	Open a socket for read/write
`ReadLine(`*maxlen*`, `*file_handle*`)`	Read a line of text from a file
`ReadChar(`*file_handle*`)`	Read a character from a file
`WriteLine(`*string*`, `*file_handle*`)`	Write a line of text to a file

`WriteChar(`*c*`,` *file_handle*`)`	Write a character to a file
`TempFile(`*prefix*`)`	Create a temporary file name
`ListDirectory(`*path*`,` *filter*`)`	Return handle to list of file names
`MakeDir(`*path*`)`	Create directory tree
`FileStat(`*path*`,` *array*`)`	Get file/directory statistics
`DeleteFile(`*path*`)`	Destroy file or empty directory
`MoveFile(`*from_path*`,` *to_path*`)`	Move (rename) file
`CopyFile(`*from_path*`,` *to_path*`)`	Copy file
`CreateBak(`*path*`)`	Move file to backup
`Md5Digest(`*path*`)`	Return file digest string
Socket and *Xic* Client/Server Interface	
`ReadData(`*size*`,` *skt_handle*`)`	Read data from a socket
`ReadReply(`*retcode*`,` *skt_handle*`)`	Read a message from the *Xic* server
`ConvertReply(`*message*`,` *retcode*`)`	Parse *Xic* server response
`WriteMsg(`*string*`,` *skt_handle*`)`	Write a message to a socket
System Command Interface	
`Shell(`*command*`)`	Execute a shell command
`System(`*command*`)`	Execute a shell command
`GetPID(`*parent*`)`	Return process ID
Menu Buttons	
`SetButtonStatus(`*menu*`,` *button*`,` *set*`)`	Set button toggle status
`GetButtonStatus(`*menu*`,` *button*`)`	Return button toggle status
`PressButton(`*menu*`,` *button*`)`	Synthesize a button press
`BtnDown(`*num*`,` *state*`,` *x*`,` *y*`,` *widget*`)`	Synthesize a button press
`BtnUp(`*num*`,` *state*`,` *x*`,` *y*`,` *widget*`)`	Synthesize a button release
`KeyDown(`*keysym*`,` *state*`,` *widget*`)`	Synthesize a key press
`KeyUp(`*keysym*`,` *state*`,` *widget*`)`	Synthesize a key release
Mouse Input	
`Point(`*array*`)`	Wait for a mouse button press
`Selection()`	Wait for key press, allow selections
Graphical Input	
`PopUpInput(`*message*`,` *default*`,` *buttontext*`,` *multiline*`)`	Pop up text input dialog
`PopUpAffirm(`*message*`)`	Pop up yes/no dialog
`PopUpNumeric(`*message*`,` *initval*`,` *minval*`,` *maxval*`,` *delta*`,` *numdgt*`)`	Pop up numeric entry dialog
Text Input	
`AskReal(`*prompt*`,` *default*`)`	Prompt for a number from prompt line
`AskString(`*prompt*`,` *default*`)`	Prompt for a string from prompt line
`AskConsoleReal(`*prompt*`,` *default*`)`	Prompt for a number from console
`AskConsoleString(`*prompt*`,` *default*`)`	Prompt for a string from console
`GetKey()`	Wait for key press
Text Output	
`SepString(`*string*`,` *repeat*`)`	Create separation or indentation string
`ShowPrompt(`*arg_list*`)`	Show arguments on prompt line
`SetIndent(`*level*`)`	Set indentation level for printing
`SetPrintLimits(`*num_array_elts*`,` *num_zoids*`)`	Limit number of array values and trapezoids printed

`Print(`*arg_list*`)`	Print arguments to console window
`PrintLog(`*file_handle*`, `*arg_list*`)`	Print arguments to file
`PrintString(`*arg_list*`)`	Print arguments to a string
`PrintStringEsc(`*arg_list*`)`	Print arguments to a string
`Message(`*arg_list*`)`	Print arguments to pop-up window
`ErrorMsg(`*arg_list*`)`	Print arguments to pop-up error window
`TextWindow(`*fname*`, `*readonly*`)`	Show file in text editor

The third group of main module functions:

Main Functions 3	
Grid and Edge Snapping	
`SetMfgGrid(`*mfg_grid*`)`	Set the manufacturing grid
`GetMfgGrid()`	Return the manufacturing grid
`SetGrid(`*interval*`, `*snap*`, `*win*`)`	Set grid parameters for window
`GetGridInterval(`*win*`)`	Return fine grid spacing
`GetSnapInterval(`*win*`)`	Return the snap grid spacing
`GetGridSnap(`*win*`)`	Return grid snap number
`ClipToGrid(`*coord*`, `*win*`)`	Move coord to grid
`SetEdgeSnappingMode(`*win*`, `*mode*`)`	Set edge snapping scope for window
`SetEdgeOffGrid(`*win*`, `*off_grid*`)`	Enable off-grid edge snapping in window
`SetEdgeNonManh(`*win*`, `*non_manh*`)`	Enable non-Manhattan edge snapping in window
`SetEdgeWireEdge(`*win*`, `*wire_edge*`)`	Snap to wire edges in window
`SetEdgeWirePath(`*win*`, `*wire_path*`)`	Snap to wire path in window
`GetEdgeSnappingMode(`*win*`)`	Return edge snapping mode for windoiw
`GetEdgeOffGrid(`*win*`)`	Return off-grid edge snapping flag for window
`GetEdgeNonManh(`*win*`)`	Return non-Manhattan edge snapping flag for window
`GetEdgeWireEdge(`*win*`)`	Return wire edge snapping flag for window
`GetEdgeWirePath(`*win*`)`	Return wire path snapping flag for window
`SetRulerSnapToGrid(`*snap*`)`	Set ruler command grid snapping state
`SetRulerEdgeSnappingMode(`*mode*`)`	Set ruler command edge snapping mode
`SetRulerEdgeOffGrid(`*off_grid*`)`	Set ruler command edge snapping off-grid state
`SetRulerEdgeNonManh(`*non_manh*`)`	Set ruler command edge snapping non-Manhattan state
`SetRulerEdgeWireEdge(`*wire_edge*`)`	Set ruler command edge snapping wire-edge state
`SetRulerEdgeWirePath(`*wire_path*`)`	Set ruler command edge snapping wire-path state
`GetRulerSnapToGrid()`	Return ruler command grid snapping state
`GetRulerEdgeSnappingMode()`	Return ruler command edge snapping mode
`GetRulerEdgeOffGrid()`	Return ruler command edge snapping off-grid state
`GetRulerEdgeNonManh()`	Return ruler command edge snapping non-Manhattan state
`GetRulerEdgeWireEdge()`	Return ruler command edge snapping wire-edge state

`GetRulerEdgeWirePath()`	Return ruler command edge snapping wire-path state
Grid Style	
`ShowGrid(`*on*`, `*win*`)`	Set grid visibility in window
`ShowAxes(`*style*`, `*win*`)`	Set axes style in window
`SetGridStyle(`*style*`, `*win*`)`	Set grid line style
`GetGridStyle(`*win*`)`	Return grid line style
`SetGridCrossSize(`*xsize*`, `*win*`)`	Set grid "dot" cross size
`GetGridCrossSize(`*win*`)`	Return grid "dot" cross size
`SetGridOnTop(`*ontop*`, `*win*`)`	Set grid on top of geometry
`GetGridOnTop(`*win*`)`	Return grid top/bottom status
`SetGridCoarseMult(`*mult*`, `*win*`)`	Set coarse grid spacing multiple
`GetGridCoarseMult(`*win*`)`	Return coarse grid spacing multiple
`SaveGrid(`*regnum*`, `*win*`)`	Save grid parameters in register
`RecallGrid(`*regnum*`, `*win*`)`	Recall grid parameters from register
Current Layer	
`GetCurLayer()`	Return name of current layer
`GetCurLayerIndex()`	Return index of current layer
`SetCurLayer(`*name*`)`	Set current layer, layer must exist
`SetCurLayerFast(`*name*`)`	As SetCurLayer, but no screen update
`NewCurLayer(`*name*`)`	Set current layer, create if necessary
`GetCurLayerAlias()`	Return alias name of current layer
`SetCurLayerAlias(`*alias*`)`	Set alias name of current layer
`GetCurLayerDescr()`	Return description of current layer
`SetCurLayerDescr(`*descr*`)`	Set description of current layer
Layer Table	
`LayersUsed()`	Return number of layers in table
`AddLayer(`*name*`, `*index*`)`	Add a new layer
`RemoveLayer(`*stdlyr*`)`	Remove a layer
`RenameLayer(`*oldname*`, `*newname*`)`	Give a new name to a layer
`LayerHandle(`*down*`)`	Return a handle to a list of layer names
`GenLayers(`*stringlist_handle*`)`	Return a layer name and advance list to next
`GetLayerPalette(`*regnum*`)`	Return list of palette layers
`SetLayerPalette(`*list*`, `*regnum*`)`	Save list of palette layers
Layer Database	
`GetLayerNum(`*name*`)`	Return component layer number for name
`GetLayerName(`*num*`)`	Return component layer name for number
`IsPurposeDefined(`*name*`)`	Return true if name matches a purpose
`GetPurposeNum(`*name*`)`	Return purpose number for name
`GetPurposeName(`*num*`)`	Return purpose name for number
Layers	
`GetLayerLayerNum(`*stdlyr*`)`	Return the component layer number for layer
`GetLayerPurposeNum(`*stdlyr*`)`	Return the purpose number for layer
`GetLayerAlias(`*stdlyr*`)`	Return the alias for layer
`SetLayerAlias(`*stdlyr*`, `*alias*`)`	Set the alias for layer
`GetLayerDescr(`*stdlyr*`)`	Return the description for layer
`SetLayerDescr(`*stdlyr*`, `*descr*`)`	Set the description for layer
`IsLayerDefined(`*lname*`)`	Return nonzero if layer exists with given name

`IsLayerVisible(`*stdlyr*`)`	Return true if layer is visible
`SetLayerVisible(`*stdlyr*`, `*visible*`)`	Set layer visibility flag
`IsLayerSelectable(`*stdlyr*`)`	Return true if layer is selectable
`SetLayerSelectable(`*stdlyr*`, `*selectable*`)`	Set layer selectability flag
`IsLayerSymbolic(`*stdlyr*`)`	Return true if layer is symbolic
`SetLayerSymbolic(`*stdlyr*`, `*symbolic*`)`	Set layer symbolic flag
`IsLayerNoMerge(`*stdlyr*`)`	Return true if layer has no_merge set
`SetLayerNoMerge(`*stdlyr*`, `*nomerge*`)`	Set layer no_merge flag
`GetLayerMinDimension(`*stdlyr*`)`	Return minimum dimension
`GetLayerWireWidth(`*stdlyr*`)`	Return default wire width
`AddLayerGdsOutMap(`*stdlyr*`, `*layer_num*`, `*datatype*`)`	Add GDSII output layer mapping
`RemoveLayerGdsOutMap(`*stdlyr*`, `*layer_num*`, `*datatype*`)`	Remove GDSII output layer mapping
`AddLayerGdsInMap(`*stdlyr*`, `*string*`)`	Add GDSII input layer mapping
`ClearLayerGdsInMap(`*stdlyr*`)`	Clear GDSII input layer mapping
`SetLayerNoDRCdatatype(`*stdlyr*`, `*datatype*`)`	Set GDSII NoDRC datatype
Layers – Extraction Support	
`SetLayerExKeyword(`*stdlyr*`, `*string*`)`	Set extraction keyword/value of layer
`SetCurLayerExKeyword(`*string*`)`	Set extraction keyword/value of current layer
`RemoveLayerExKeyword(`*stdlyr*`, `*keyword*`)`	Remove extraction keyword spec from layer
`RemoveCurLayerExKeyword(`*keyword*`)`	Remove extraction keyword spec from current layer
`IsLayerConductor(`*stdlyr*`)`	Return nonzero for Conductor
`IsLayerRouting(`*stdlyr*`)`	Return nonzero for Routing
`IsLayerGround(`*stdlyr*`)`	Return nonzero for GroundPlane
`IsLayerContact(`*stdlyr*`)`	Return nonzero for Contact
`IsLayerVia(`*stdlyr*`)`	Return nonzero for Via
`IsLayerViaCut(`*stdlyr*`)`	Return nonzero for ViaCut
`IsLayerDielectric(`*stdlyr*`)`	Return nonzero for Dielectric
`IsLayerDarkField(`*stdlyr*`)`	Return nonzero for DarkField
`GetLayerThickness(`*stdlyr*`)`	Return Thickness
`GetLayerRho(`*stdlyr*`)`	Return resistivity
`GetLayerResis(`*stdlyr*`)`	Return resistance per square
`GetLayerTau(`*stdlyr*`)`	Return Drude relaxation time
`GetLayerEps(`*stdlyr*`)`	Return dielectric constant
`GetLayerCap(`*stdlyr*`)`	Return capacitance per area
`GetLayerCapPerim(`*stdlyr*`)`	Return capacitance per length
`GetLayerLambda(`*stdlyr*`)`	Return penetration depth
Selections	
`SetLayerSpecific(`*state*`)`	Restrict selectability to current layer
`SetLayerSearchUp(`*state*`)`	Set layer traversal direction
`SetSelectMode(`*ptr_mode*`, `*area_mode*`, `*sel_mode*`)`	Set selection modes
`SetSelectTypes(`*string*`)`	Set selectable object types
`Select(`*left*`, `*bottom*`, `*right*`, `*top*`, `*types*`)`	Select objects
`Deselect()`	Deselect objects
Pseudo-Flat Generator	

`FlatObjList(`*l*`,` *b*`,` *r*`,` *t*`,` *depth*`)`	Return list of object copies
`FlatObjGen(`*l*`,` *b*`,` *r*`,` *t*`,` *depth*`)`	Return handle to object generator
`FlatObjGenLayers(`*l*`,` *b*`,` *r*`,` *t*`,` *depth*`,` *layers*`)`	Return handle to object generator
`FlatGenNext(`*handle*`)`	Return handle to next object copy
`FlatGenCount(`*handle*`)`	Count objects accessible by handle
`FlatOverlapList(`*object_handle*`,` *touch_ok*`,` *depth*`,` *layers*`)`	Return handle to next object copy
Geometry Measurement	
`Distance(`*x*`,` *y*`,` *x1*`,` *y1*`)`	Measure distance between points
`MinDistPointToSeg(`*x*`,` *y*`,` *x1*`,` *y1*`,` *x2*`,` *y2*`,` *aret*`)`	Measure minimum distance between point and line segment
`MinDistPointToObj(`*x*`,` *y*`,` *object_handle*`,` *aret*`)`	Measure minimum distance between point and object
`MinDistSegToObj(`*x1*`,` *y1*`,` *x2*`,` *y2*`,` *object_handle*`,` *aret*`)`	Measure minimum distance between line segment and object
`MinDistObjToObj(`*object_handle1*`,` *object_handle2*`,` *aret*`)`	Measure minimum distance between objects
`MaxDistPointToObj(`*x*`,` *y*`,` *object_handle*`,` *aret*`)`	Measure maximum distance from point to object
`MaxDistObjToObj(`*object_handle1*`,` *object_handle2*`,` *aret*`)`	Measure maximum distance between objects
`Intersect(`*object_handle1*`,` *object_handle2*`,` *touchok*`)`	Check if objects touch or overlap

Functions related to reading and writing of layout data:

Layout File Input/Output Functions	
Layer Conversion Aliasing	
`ReadLayerCvAliases(`*handle_or_filename*`)`	Read file containing layer conversion aliases
`DumpLayerCvAliases(`*handle_or_filename*`)`	Dump file containing layer conversion aliases
`ClearLayerCvAliases()`	Delete all layer conversion aliases
`AddLayerCvAlias(`*lname*`,` *new_lname*`)`	Add layer conversion alias to table
`RemoveLayerCvAlias(`*lname*`)`	Remove layer conversion alias from table
`GetLayerCvAlias(`*lname*`)`	Return conversion alias for layer name
Cell Name Mapping	
`SetMapToLower(`*state*`,` *rw*`)`	Set cell name case conversion
`SetMapToUpper(`*state*`,` *rw*`)`	Set cell name case conversion
Cell Table	
`CellTabAdd(`*cellname*`,` *expand*`)`	Add cell(s) to cell table
`CellTabCheck(`*cellname*`)`	Return true if name is in cell table
`CellTabRemove(`*cellname*`)`	Remove name from cell table
`CellTabList(`*cellname*`)`	List names in cell table
`CellTabClear(`*cellname*`)`	Clear all names from cell table
Windowing and Flattening	
`SetConvertFlags(`*use_window*`,` *clip*`,` *flatten*`,` *ecf_level*`,` *rw*`)`	Set modes for format translation or output
`SetConvertArea(`*l*`,` *b*`,` *r*`,` *t*`,` *rw*`)`	Set filter/clipping area for translation or output
Scale Factor	
`SetConvertScale(`*scale*`,` *which*`)`	Set scale factor for import/export

Export Flags	
SetStripForExport(*state*)	Set flag to write physical data only
SetSkipInvisLayers(*code*)	Set code to skip invisible layers in output
Import Flags	
SetMergeInRead(*state*)	Enable box and wire merging in input
Layout File Format Conversion	
FromArchive(*file_or_chd*, *destination*)	Translate archive file to another format
FromTxt(*text_file*, *gds_file*)	Create GDSII file from GDSII text
FromNative(*dir_path*, *archive_file*)	Translate native cell files to archive
Export Layout File	
SaveCellAsNative(*cellname*, *directory*)	Write a native cell file in the directory
Export(*filepath*, *allcells*)	Write data to disk
ToXIC(*destination_dir*)	Write *Xic* files
ToCGX(*cgx_name*)	Write CGX file
ToCIF(*cif_name*)	Write CIF file
ToGDS(*gds_name*)	Write GDSII file
ToGdsLibrary(*gds_name*, *cellname_list*)	Write GDSII library file
ToOASIS(*oas_name*)	Write OASIS file
ToTxt(*archive_file*, *text_file*, *cmdargs*)	Write text-mode GDSII/CGX/OASIS file
Cell Hierarchy Digest	
FileInfo(*filename*, *handle_or_filename*, *flags*)	Obtain info about archive file
OpebCellHierDigest(*filename*, *info_saved*)	Create new CHD
WriteCellHierDigest(*chd_name*, *filename*, *incl_geom*, *no_compr*)	Write CHD to file
ReadCellHierDigest(*filename*, *cgd_type*)	Obtain CHD from file
ChdList()	Return a list of CHD access names
ChdChangeName(*old_chd_name*, *new_chd_name*))	Change the access name of a CHD
ChdIsValid(*chd_name*)	Return true if named CHD exists
ChdDestroy(*chd_name*)	Destroy the CHD
ChdInfo(*chd_name*, *handle_or_filename*, *flags*)	Obtain CHD information
ChdFileName(*chd_name*)	Obtain archive file name
ChdFileType(*chd_name*)	Obtain archive file format
ChdTopCells(*chd_name*)	Obtain archive top-level cell names
ChdListCells(*chd_name*, *cellname*, *mode*, *all*)	Obtain list of cell names
ChdLayers(*chd_name*)	Obtain layers used in archive
ChdInfoMode(*chd_name*)	Return saved info mode
ChdInfoLayers(*chd_name*, *cellname*)	Return saved layer info
ChdInfoCells(*chd_name*)	Return saved cell names
ChdInfoCounts(*chd_name*)	Return saved statistics
ChdCellBB(*chd_name*, *cellname*, *array*)	Obtain cell bounding box
ChdSetDefCellName(*chd_name*, *cellname*)	Configure default cell name
ChdDefCellName(*chd_name*)	Obtain default cell name
ChdLoadGeometry(*chd_name*)	Create and link to a new Cell Geometry Digest
ChdLinkCgd(*chd_name*, *cgd_name*)	Link or unlink a CGD to the CHD
ChdGetGeomName(*chd_name*)	Return name of attached Cell Geometry Digest
ChdClearGeometry(*chd_name*)	Unlink attached Cell Geometry Digest

ChdSetSkipFlag(*chd_name*, *cellname*, *skip*)	Set or clear skip flag
ChdClearSkipFlags(*chd_name*)	Clear all skip flags
ChdCompare(*chd_name1*, *cname1*, *chd_name2*, *cname2*, *layer_list*, *skip_layers*, *maxdiffs*, *obj_types*, *geometric*, *array*)	Compare objects in cells
ChdCompareFlat(*chd_name1*, *cname1*, *chd_name2*, *cname2*, *layer_list*, *skip_layers*, *maxdiffs*, *area*, *coarse_mult*, *find_grid*, *array*)	Compare objects in flat cell hierarchies
ChdEdit(*chd_name*, *scale*, *cellname*)	Open cell for editing
ChdOpenFlat(*chd_name*, *scale*, *cellname*, *array*, *clip*)	Read a flattened hierarchy into memory
ChdSetFlatReadTransform(*tfstring*, *x*, *y*)	Set a transform for flat reading
ChdEstFlatMemoryUse(*chd_name*, *cellname*, *array*, *counts_array*)	Estimate memory required for flat read
ChdWrite(*chd_name*, *scale*, *cellname*, *array*, *clip*, *all*, *flatten*, *ecf_level*, *outfile*)	Write cells to file
ChdWriteSplit(*chd_name*, *cellname*, *basename*, *array*, *regions_or_gridsize*, *numregions_or_bloatval*, *maxdepth*, *scale*, *flags*)	Write to flat files
ChdCreateReferenceCell(*chd_name*, *cellname*)	Create a reference cell in memory
ChdLoadCell(*chd_name*, *cellname*)	Load cell in memory, reference subcells
ChdIterateOverRegion(*chd_name*, *cellname*, *funcname*, *array*, *coarse_mult*, *fine_grid*, *bloat_val*)	Iterate over grid, call callback function
ChdWriteDensityMaps(*chd_name*, *cellname*, *array*, *coarse_mult*, *fine_grid*, *bloat*, *save*)	Iterate over grid, compute density
Cell Geometry Digest	
OpenCellGeomDigest(*idname*, *string*, *type*)	Create a new CGD
NewCellGeomDigest()	Create a new empty CGD
WriteCellGeomDigest(*cgd_name*, *filename*)	Write CGD to file
CgdList()	Return a list of CGD access names
CgdChangeName(*old_cgd_name*, *new_cgd_name*)	Change the access name of a CGD
CgdIsValid(*cgd_name*)	Return true if named CGD exists
CgdDestroy(*cgd_name*)	Destroy the CGD
CgdIsValidCell(*cgd_name*, *cellname*)	Return true if cell is found in CGD
CgdIsValidLayer(*cgd_name*, *cellname*, *layername*)	Return true if cell containing layer is found in CGD
CgdRemoveCell(*cgd_name*, *cellname*)	Remove a cell from the CGD
CgdIsCellRemoved(*cgd_name*, *cellname*)	Return true if the cell was removed from the CGD
CgdRemoveLayer(*cgd_name*, *cellname*, *layername*)	Remove layer data from a cell in the CGD
CgdAddCells(*cgd_name*, *chd_name*, *cells_list*)	Add cells to the CGD
CgdContents(*cgd_name*, *cellname*, *layername*)	List contents of CGD
CgdOpenGeomStream(*cgd_name*, *cellname*, *layername*)	Open geometry stream from CGD
GsReadObject(*gs_handle*)	Read geometry from a geometry stream

`GsDumpOasisText(`*gs_handle*`)`	Dump OASIS ASCII text representation to console
Assembly Stream	
`StreamOpen(`*outfile*`)`	Open an assembly stream
`StreamTopCell(`*stream_handle*`,` *cellname*`)`	Define a top-level cell in the stream
`StreamSource(`*stream_handle*`,` *file_or_chd*`,` *scale*`,` *layer_filter*`,` *name_change*`)`	Register a source archive for streaming
`StreamInstance(`*stream_handle*`,` *cellname*`,` *x*`,` *y*`,` *my*`,` *rot*`,` *magn*`,` *scale*`,` *no_hier*`,` *ecf_level*`,` *flatten*`,` *array*`,` *clip*`)`	Add an instance conversion spec to a source
`StreamRun(`*stream_handle*`)`	Initiate streaming to output

First group of functions for geometry editing

Geometry Editing Functions 1	
General Editing	
`ClearCell(`*undoable*`,` *layer_list*`)`	Clear content of current cell
`Commit()`	Finalize changes in database
`Undo()`	Undo last operation
`Redo()`	Redo last undone operation
`SelectLast(`*types*`)`	Select most recent new object
Current Transform	
`SetTransform(`*angle_or_string*`,` *reflection*`,` *magnification*`)`	Set current transform
`StoreTransform(`*register*`)`	Save current transform parameters
`RecallTransform(`*register*`)`	Recall current transform parameters
`GetTransformString()`	Return acode string for the current transform
`GetCurAngle()`	Return current transform angle
`GetCurMX()`	Return current transform mirror-x
`GetCurMY()`	Return current transform mirror-y
`GetCurMagn()`	Return current transform magnification
`UseTransform(`*enable*`,` *x*`,` *y*`)`	Enable use of current transform
Derived Layers	
`AddDerivedLayer(`*lname*`,` *index*`,` *lexpr*`)`	Add a derived layer definition
`RemDerivedLayer(`*lname*`)`	Remove a derived layer definition
`IsDerivedLayer(`*lname*`)`	True if name matches a derived layer definition
`GetDerivedLayerIndex(`*lname*`)`	Return the index of the specified derived layer
`GetDerivedLayerExpString(`*lname*`)`	Return the layer expression string of the specified derived layer
`GetDerivedLayerLexpr(`*lname*`,` *noexp*`)`	Return a layer expression object for the specified derived layer
`EvalDerivedLayers(`*list*`,` *array*`)`	Evaluate the list of derived layers in an area
`ClearDerivedLayers(`*list*`)`	Clear geometry of derived layers in list
Object Management by Handles	
`ListElecInstances()`	List electrical cell instances from current cell
`ListPhysInstances()`	List physical cell instances from current cell
`SelectHandle()`	Return handle to a list of selected objects
`SelectHandleTypes(`*types*`)`	Return handle to a list of selected objects of given types

`AreaHandle(`*l*`, `*b*`, `*r*`, `*t*`, `*types*`)`	Return handle to a list of objects in area
`ObjectHandleDup(`*object_handle*`, `*types*`)`	Duplicate handle with given object types
`ObjectHandlePurge(`*object_handle*`, `*types*`)`	Remove from list objects with given types
`ObjectNext(`*object_handle*`)`	Advance list to next object
`MakeObjectCopy(`*numpts*`, `*array*`)`	Create a phony object copy
`ObjectString(`*object_handle*`)`	Return CIF-like string for object
`ObjectCopyFromString(`*object_handle*`, `*layer*`)`	Return new object from CIF-like string
`FilterObjects(`*object_list*`, `*template_list*`, `*all*`, `*touchok*`, `*remove*`)`	Select objects via template
`CheckObjectsConnected(`*object_handle*`)`	Return 1 if objects in list form one group
`CheckForHoles(`*object_handle*`, `*all*`)`	Return 1 if object(s) have "holes"
`FilterObjectsA(`*object_list*`, `*array*`, `*array_size*`, `*touchok*`, `*remove*`)`	Select objects via given polygon
`BloatObjects(`*object_handle*`, `*all*`, `*dimen*`, `*lname*`, `*mode*`)`	Create list of bloated objects
`EdgeObjects(`*object_handle*`, `*all*`, `*dimen*`, `*lname*`, `*mode*`)`	Create list of edge "wire" polygons
`ManhattanizeObjects(`*object_handle*`, `*all*`, `*dimen*`, `*lname*`, `*mode*`)`	Create list of Manhattanized objects
`GroupObjects(`*object_handle*`, `*array*`)`	Create connected groups of objects
`JoinObjects(`*object_handle*`, `*lname*`)`	Join touching objects in a list
`SplitObjects(`*object_handle*`, `*all*`, `*lname*`, `*vert*`)`	Split into trapezoids objects in a list
`DeleteObjects(`*object_handle*`, `*all*`)`	Delete objects
`SelectObjects(`*object_handle*`, `*all*`)`	Select objects
`DeselectObjects(`*object_handle*`, `*all*`)`	Deselect objects
`MoveObjects(`*object_handle*`, `*all*`, `*refx*`, `*refy*`, `*x*`, `*y*`)`	Move object(s)
`MoveObjectsToLayer(`*object_handle*`, `*all*`, `*refx*`, `*refy*`, `*x*`, `*y*`, `*oldlayer*`, `*newlayer*`)`	Move object(s) with layer change
`CopyObjects(`*object_handle*`, `*all*`, `*refx*`, `*refy*`, `*x*`, `*y*`, `*repcnt*`)`	Copy object(s)
`CopyObjectsToLayer(`*object_handle*`, `*all*`, `*refx*`, `*refy*`, `*x*`, `*y*`, `*oldlayer*`, `*newlayer*`, `*repcnt*`)`	Copy object(s) with layer change
`CopyObjectsH(`*object_handle*`, `*all*`, `*refx*`, `*refy*`, `*x*`, `*y*`, `*oldlayer*`, `*newlayer*`, `*todb*`)`	Copy object(s) to handle
`GetObjectType(`*object_handle*`)`	Return the object's type code
`GetObjectID(`*object_handle*`)`	Return the object's id number
`GetObjectArea(`*object_handle*`)`	Return the object's area in square microns
`GetObjectPerim(`*object_handle*`)`	Return the object's perimeter in microns
`GetObjectCentroid(`*object_handle*`, `*array*`)`	Compute the object's centroid point
`GetObjectBB(`*object_handle*`, `*array*`)`	Return the object's bounding box
`SetObjectBB(`*object_handle*`, `*array*`)`	Set the object's bounding box, scale object
`GetObjectListBB(`*object_handle*`, `*array*`)`	Return the bounding box of all objects in list
`GetObjectXY(`*object_handle*`, `*array*`)`	Return the object's reference point
`SetObjectXY(`*object_handle*`, `*x*`, `*y*`)`	Set the object's reference point
`GetObjectLayer(`*object_handle*`)`	Return the object's layer name
`SetObjectLayer(`*object_handle*`, `*layername*`)`	Set the object's layer
`GetObjectFlags(`*object_handle*`)`	Return the object's flags

`SetObjectNoDrcFlag(`*object_handle*`,` *value*`)`	Set or unset the `NoDRC` object flag
`SetObjectMark1Flag(`*object_handle*`,` *value*`)`	Set or unset the `Mark1` object flag
`SetObjectMark2Flag(`*object_handle*`,` *value*`)`	Set or unset the `Mark2` object flag
`GetObjectState(`*object_handle*`)`	Return the object's state
`GetObjectGroup(`*object_handle*`)`	Return the object's conductor group number
`SetObjectGroup(`*object_handle*`,` *group_num*`)`	Set the object's conductor group number
`GetObjectCoords(`*object_handle*`,` *array*`)`	Return the object's coordinates
`SetObjectCoords(`*object_handle*`,` *array*`,` *size*`)`	Set the object's coordinates
`GetObjectMagn(`*object_handle*`)`	Return the magnification of a subcell
`SetObjectMagn(`*object_handle*`,` *magn*`)`	Set object's magnification, rescale object
`GetWireWidth(`*object_handle*`)`	Return width of wire
`SetWireWidth(`*object_handle*`,` *width*`)`	Set width of wire
`GetWireStyle(`*object_handle*`)`	Return wire end style
`SetWireStyle(`*object_handle*`,` *code*`)`	Set wire end style
`SetWireToPoly(`*object_handle*`)`	Convert wire to polygon
`GetWirePoly(`*object_handle*`,` *array*`)`	Return wire bounding polygon
`GetLabelText(`*object_handle*`)`	Return text of label
`SetLabelText(`*object_handle*`,` *text*`)`	Set text in label
`GetLabelFlags(`*object_handle*`)`	Return flags for label
`SetLabelFlags(`*object_handle*`,` *flags*`)`	Set flags for label
`GetInstanceArray(`*object_handle*`,` *array*`)`	Return instance array parameters
`SetInstanceArray(`*object_handle*`,` *array*`)`	Set instance array parameters, resize array
`GetInstanceXform(`*object_handle*`)`	Return instance transformation string
`GetInstanceXformA(`*object_handle*`,` *array*`)`	Return instance transformation in array
`SetInstanceXform(`*object_handle*`,` *transform*`)`	Set instance transformation from string
`SetInstanceXformA(`*object_handle*`,` *array*`)`	Set instance transformation from array
`GetInstanceMaster(`*object_handle*`)`	Return name of instance master cell
`SetInstanceMaster(`*object_handle*`,` *newname*`)`	Set instance master, replace instance
`GetInstanceName(`*object_handle*`)`	Return name of instance
`SetInstanceName(`*object_handle*`,` *newname*`)`	Set instance name property
`GetInstanceAltName(`*object_handle*`)`	Return alternate name of instance
`GetInstanceType(`*object_handle*`)`	Return instance type code
`GetInstanceIdNum(`*object_handle*`)`	Return instance id number
`GetInstanceAltIdNum(`*object_handle*`)`	Return instance alternate id number

Second group of functions for geometry editing

Geometry Editing Functions 2	
Cells, PCells, Vias, and Instance Placement	
`CheckPCellParam(`*library*`,` *cell*`,` *view*`,` *pname*`,` *value*`)`	Validate a parameter value
`CheckPCellParams(`*library*`,` *cell*`,` *view*`,` *params*`)`	Validate a parameter list
`CreateCell(`*cellname*`, [`*orig_x*`,` *orig_y*`])`	Create new cell from selected objects
`CopyCell(`*name*`,` *newname*`)`	Copy a cell
`RenameCell(`*oldname*`,` *newname*`)`	Globally rename cell in memory, fix references
`DeleteEmpties(`*recurse*`)`	Delete empty cells
`Place(`*cellname*`,` *x*`,` *y* `[,` *refpt*`,` *array*`,` *smash*`,` *usegui*`,` *tfstring*`])`	Place an instance

PlaceH(*cellname*, *x*, *y* [, *refpt*, *array*, *smash*, *usegui*, *tfstring*])	Place an instance, return handle
PlaceSetArrayParams(*nx*, *ny*, *dx*, *dy*)	Set instrance arraying parameters
PlaceSetPCellParams(*library*, *cell*, *view*, *params*)	Set pcell parameter string
Replace(*cellname*, *add_xform*, *array*)	Replace an instance
OpenViaSubMaster(*vianame*, *defnstr*)	Define a standard via variant
Clipping Functions	
ClipAround(*object_handle1*, *all1*, *object_handle2*, *all2*)	Clip object around other objects
ClipAroundCopy(*object_handle1*, *all1*, *object_handle2*, *all2*, *lname*)	Clip objects around other objects, return copies
ClipTo(*object_handle1*, *all1*, *object_handle2*, *all2*)	Clip objects to other objects
ClipToCopy(*object_handle1*, *all1*, *object_handle2*, *all2*, *lname*)	Clip objects to other objects, return copies
ClipObjects(*object_handle*, *merge*)	Clip object list so no overlap
ClipIntersectCopy(*object_handle1*, *all1*, *object_handle2*, *all2*, *lname*)	Exclusive-or objects or lists
Other Object Management Functions	
ChangeLayer()	Change layer of selected objects
Bloat(*dimen*, *mode*)	Bloat selected objects
Manhattanize(*dimen*, *mode*)	Manhattanize selected objects
Join()	Join selected objects
Decompose(*vert*)	Convert selected objects to trapezoids
Box(*left*, *bottom*, *right*, *top*)	Create a box
BoxH(*left*, *bottom*, *right*, *top*)	Create a box, return handle
Polygon(*num*, *arraypts*)	Create a polygon
PolygonH(*num*, *arraypts*)	Create a polygon, return handle
Arc(*x*, *y*, *rad1X*, *rad1Y*, *rad2X*, *rad2Y*, *ang_start*, *ang_end*)	Create an arc polygon
ArcH(*x*, *y*, *rad1X*, *rad1Y*, *rad2X*, *rad2Y*, *ang_start*, *ang_end*)	Create an arc polygon, return handle
Round(*x*, *y*, *rad*)	Create a disk polygon
RoundH(*x*, *y*, *rad*)	Create a disk polygon, return handle
HalfRound(*x*, *y*, *rad*, *dir*)	Create a half-disk polygon
HalfRoundH(*x*, *y*, *rad*, *dir*)	Create a half-disk polygon, return handle
Sides(*numsides*)	Set the number of sides used for round objects
Wire(*width*, *num*, *arraypts*, *end_style*)	Create a wire
WireH(*width*, *num*, *arraypts*, *end_style*)	Create a wire, return handle
Label(*text*, *x*, *y* [, *width*, *height*, *flags*])	Create a label
LabelH(*text*, *x*, *y* [, *width*, *height*, *flags*])	Create a label, return handle
Logo(*string*, *x*, *y* [, *width*, *height*])	Create physical text
Justify(*hj*, *vj*)	Set default text justification
Delete()	Delete selected objects
Erase(*left*, *bottom*, *right*, *top*)	Erase objects in area
EraseUnder()	Erase overlap with selected objects
Yank(*left*, *bottom*, *right*, *top*)	Grab geometry into buffer
Put(*x*, *y*, *bufnum*)	Place stored geometry

`Xor(`*left*`, `*bottom*`, `*right*`, `*top*`)`	Exclusive-or geometry in area
`Copy(`*fromx*`, `*fromy*`, `*tox*`, `*toy*`, `*repcnt*`)`	Copy selected objects
`CopyToLayer(`*fromx*`, `*fromy*`, `*tox*`, `*toy*`, `*oldlayer*`, `*newlayer*`, `*repcnt*`)`	Copy selected objects and change layer
`Move(`*fromx*`, `*fromy*`, `*tox*`, `*toy*`)`	Move selected objects
`MoveToLayer(`*fromx*`, `*fromy*`, `*tox*`, `*toy*`, `*oldlayer*`, `*newlayer*`)`	Move selected objects and change layer
`Rotate(`*x*`, `*y*`, `*ang*`, `*remove*`)`	Rotate selected objects
`RotateToLayer(`*x*`, `*y*`, `*ang*`, `*oldlayer*`, `*newlayer*`, `*remove*`)`	Rotate selected objects and change layer
`Split(`*x*`, `*y*`, `*flag*`, `*orient*`)`	Divide selected objects
`Flatten(`*depth*`, `*use_merge*`, `*fast_mode*`)`	Flatten hierarchy
`Layer(`*string*`, `*mode*`, `*depth*`, `*recurse*`, `*noclear*`, `*use_merge*`, `*fast_mode*`)`	Apply geometric manipulations
Property Management	
`PrpHandle(`*object_handle*`)`	Return handle to a list of the object's properties
`GetPrpHandle(`*number*`)`	Return a handle to certain properties
`CellPrpHandle()`	Return handle to a list of all current cell properties
`GetCellPrpHandle(`*number*`)`	Return handle to a list of specific current cell properties
`PrpNext(`*prpty_handle*`)`	Advance to the next property
`PrpNumber(`*prpty_handle*`)`	Return the property number
`PrpString(`*prpty_handle*`)`	Return the property string
`PrptyString(`*obj_or_prp_handle*`, `*number*`)`	Return the property string
`GetPropertyString(`*number*`)`	Return property string from selected object
`GetCellPropertyString(`*number*`)`	Return property string from current cell
`PrptyAdd(`*object_handle*`, `*number*`, `*string*`)`	Add a property
`AddProperty(`*number*`, `*string*`)`	Add properties to selected objects
`AddCellProperty(`*number*`, `*string*`)`	Add property to current cell
`PrptyRemove(`*object_handle*`, `*number*`, `*string*`)`	Remove a property
`RemoveProperty(`*number*`, `*string*`)`	Remove properties from selected objects
`RemoveCellProperty(`*number*`, `*string*`)`	Remove properties from current cell

These are the computational geometry functions:

Computational Geometry and layer Expressions	
Trapezoid lists and Layer Expressions	
`SetZref(`*arg*`)`	Set background clipping zoidlist
`GetZref()`	Return background clipping zoidlist
`GetZrefBB(`*array*`)`	Return background clipping zoidlist bounding box
`AdvanceZref(`*clear*`, `*array*`)`	Establish or advance grid clipping area
`Zhead(`*zoidlist*`)`	Extract and return leading trapezoid
`Zvalues(`*zoidlist*`, `*array*`)`	Extract parameters of leading trapezoid
`Zlength(`*zoidlist*`)`	Return number of trapezoids in list
`Zarea(`*zoidlist*`)`	Return total area of trapezoids in list
`GetZlist(`*layersrc*`, `*depth*`)`	Create zoidlist from cell
`GetSqZlist(`*layername*`)`	Create zoidlist from selected objects

`TransformZ(`*zoidlist*`,` *refx*`,` *refy*`,` *newx*`,` *newy*`)`	Apply a transformation to a zoidlist
`BloatZ(`*dimen*`,` *zoidlist*`,` *mode*`)`	Bloat a zoidlist
`ExtentZ(`*zoidlist*`)`	Find the bounding box of a zoidlist
`EdgesZ(`*dimen*`,` *zoidlist*`,` *mode*`)`	Create an edge zoidlist
`ManhattanizeZ(`*dimen*`,` *zoidlist*`,` *mode*`)`	Manhattanize a zoidlist
`RepartitionZ(`*zoidlist*`)`	Canonicalize for horizontal split
`BoxZ(`*l*`,` *b*`,` *r*`,` *t*`)`	Create zoidlist from box
`ZoidZ(`*xll*`,` *xlr*`,` *yl*`,` *xul*`,` *xur*`,` *yu*`)`	Create zoidlist from trapezoid
`ObjectZ(`*object_handle*`,` *all*`)`	Create zoidlist from object(s)
`ParseLayerExpr(`*string*`)`	Create layer_expr from string
`EvalLayerExpr(`*layer_expr*`,` *zoidlist*`,` *depth*`,` *isclear*`)`	Evaluate layer expression in zoidlist
`TestCoverageFull(`*layer_expr*`,` *zoidlist*`,` *minsize*`)`	Test layer expression for full coverage of zoidlist
`TestCoveragePartial(`*layer_expr*`,` *zoidlist*`,` *minsize*`)`	Test layer expression for partial coverage of zoidlist
`TestCoverageNone(`*layer_expr*`,` *zoidlist*`,` *minsize*`)`	Test layer expression for no coverage of zoidlist
`TestCoverage(`*layer_expr*`,` *zoidlist*`,` *testfull*`)`	Test layer expression in zoidlist
`ZtoObjects(`*zoidlist*`,` *lname*`,` *join*`,` *to_dbase*`)`	Create objects from zoidlist
`ZtoTempLayer(`*longname*`,` *zoidlist*`,` *join*`)`	Put objects from zoidlist in layer
`ClearTempLayer(`*longname*`)`	Clear objects in layer
`ZtoFile(`*filename*`,` *zoidlist*`,` *ascii*`)`	Save trapezoid list in file
`ZfromFile(`*filename*`)`	Extract trapezoid list from file
`ReadZfile(`*filename*`)`	Read trapezoids from file into current cell
`ChdGetZlist(`*chd_name*`,` *cellname*`,` *scale*`,` *array*`,` *clip*`,` *all*`)`	Extract trapezoid list through CHD
Operations	
`Filt(`*zoids*`,` *lexpr*`)`	Trapezoid filtering
`GeomAnd(`*zoids1* `[,` *zoids2*`])`	Geometrical AND function
`GeomAndNot(`*zoids1*`,` *zoids2*`)`	Clip second list from first
`GeomCat(`*zoids1*`, ...)`	Concatenate zoidlists
`GeomNot(`*zoids1*`)`	Invert zoidlist
`GeomOr(`*zoids1*`, ...)`	Merge zoidlist
`GeomXor(`*zoids1* `[,` *zoids2*`])`	Exclusive-Or zoidlists
Spatial Parameter Tables	
`ReadSPtable(`*filename*`)`	Create or replace a table
`NewSPtable(`*name*`,` *x0*`,` *dx*`,` *nx*`,` *y0*`,` *dy*`,` *ny*`)`	Create a table
`WriteSPtable(`*name*`,` *filename*`)`	Write a table to a file
`ClearSPtable(`*name*`)`	Destroy a table
`FindSPtable(`*name*`,` *array*`)`	Find a table
`GetSPdata(`*name*`,` *x*`,` *y*`)`	Obtain value from table
`SetSPdata(`*name*`,` *x*`,` *y*`,` *value*`)`	Set table value
Polymorphic Flat Database	
`ChdOpenOdb(`*chd_name*`,` *scale*`,` *cellname*`,` *array*`,` *clip*`,` *dbname*`)`	Open a flat object database

`ChdOpenZdb(`*chd_name*`, `*scale*`, `*cellname*`, `*array*`, `*clip*`, `*dbname*`)`	Open a flat trapezoid database
`ChdOpenZbdb(`*chd_name*`, `*scale*`, `*cellname*`, `*array*`, `*dbname*`, `*dx*`, `*dy*`, `*bx*`, `*by*`)`	Open a binned flat trapezoid database
`GetObjectsOdb(`*dbname*`, `*layer_list*`, `*array*`)`	Read objects from database
`ListLayersDb(`*dbname*`)`	List the layers used in the database
`GetZlistDb(`*dbname*`, `*layer_name*`, `*zoidlist*`)`	Read trapezoids from database
`GetZlistZbdb(`*dbname*`, `*layer_name*`, `*nx*`, `*ny*`)`	Read trapezoids from ZBDB database
`DestroyDb(`*dbname*`)`	Destroy a database
`ShowDb(`*dbname*`, `*array*`)`	Display database region
Named String Tables	
`FindNameTable(`*tabname*`, `*create*`)`	Verify existence of or create named string table
`RemoveNameTable(`*tabname*`)`	Destroy named string table
`ListNameTables()`	List existing named string tables
`ClearNameTables()`	Destroy all named string tables
`AddNameToTable(`*tabname*`, `*name*`, `*value*`)`	Add name/value to named string table
`RemoveNameFromTable(`*tabname*`, `*name*`)`	Remove name from named string table
`FindNameInTable(`*tabname*`, `*name*`)`	Return value for name in named string table
`ListNamesInTable(`*tabname*`)`	Return list of names in named string table

These functions are specific to design rule checking:

Design Rule Checking Functions	
DRC	
`DRCstate(`*state*`)`	Set interactive DRC
`DRCsetLimits(`*batch_cnt*`, `*intr_cnt*`, `*intr_time*`, `*skip_cells*`)`	Set DRC limit values
`DRCgetLimits(`*array*`)`	Return DRC limit values
`DRCsetMaxErrors(`*value*`)`	Set the batch mode error limit
`DRCgetMaxErrors()`	Return the batch mode error limit
`DRCsetInterMaxObjs(`*value*`)`	Set the interactive mode object count limit
`DRCgetInterMaxObjs()`	Return the interactive mode object count limit
`DRCsetInterMaxTime(`*value*`)`	Set the interactive mode time limit
`DRCgetInterMaxTime()`	Return the interactive mode time limit
`DRCsetInterMaxErrors(`*value*`)`	Set the interactive mode error count limit
`DRCgetInterMaxErrors()`	Return the interactive mode error count limit
`DRCsetInterSkipInst(`*value*`)`	Set the interactive mode instance skip flag
`DRCgetInterSkipInst()`	Return the interactive mode instance skip flag
`DRCsetLevel(`*level*`)`	Set DRC error reporting level
`DRCgetLevel()`	Return DRC error reporting level
`DRCcheckArea(`*array*`, `*file_handle_or_name*`)`	Perform DRC in area
`DRCchdCheckArea(`*chdname*`, `*cellname*`, `*gridsize*`, `*array*`, `*file_handle_or_name*`, `*flatten*`)`	Perform DRC in area using CHD
`DRCcheckObjects(`*file_handle*`)`	Perform DRC for selected objects
`DRCregisterExpr(`*expr*`)`	Register a layer expression
`DRCtestBox(`*left*`, `*bottom*`, `*right*`, `*top*`, `*ld*`)`	Perform DRC for given box
`DRCtestPoly(`*num*`, `*points*`, `*ld*`)`	Perform DRC for given polygon
`DRCzList(`*layername*`, `*rulename*`, `*index*`, `*source*`)`	Create test areas in returned trapezoid list

DRCzListEx(*source*, *target*, *inside*, *outside*, *incode*, *outcode*, *dimen*)	Create test areas in returned trapezoid list

Functions specifically for the extraction system:

Extraction Functions	
Menu Commands	
DumpPhysNetlist(*filename*, *depth*, *modestring*, *names*)	Dump physical netlist
DumpElecNetlist(*filename*, *depth*, *modestring*, *names*)	Dump electrical netlist
SourceSpice(*filename*, *modestring*)	Update electrical from SPICE file
ExtractAndSet(*depth*, *modestring*)	Update electrical from physical
FindPath(*x*, *y*, *depth*, *use_extract*)	Return objects in netlist
FindPathOfGroup(*groupnum*, *depth*)	Return objects in netlist
Terminals	
ListTerminals()	List cell contact terminals
FindTerminal(*name*, *index*, *use_e*, *xe*, *ye*, *use_p*, *xp*, *yp*)	Find a cell connection terminal
CreateTerminal(*name*, *x*, *y*, *termtype*)	Create new contact terminal
DestroyTerminal(*thandle*)	Remove and destroy cell contact terminal
GetTerminalName(*thandle*)	Return terminal name
SetTerminalName(*thandle*, *name*)	Assign terminal name
GetTerminalType(*thandle*)	Return terminal type code
SetTerminalType(*thandle*, *termtype*)	Set terminal type
GetTerminalFlags(*thandle*)	Return terminal flags
SetTerminalFlags(*thandle*, *flags*)	Set terminal flags
UnsetTerminalFlags(*thandle*, *flags*)	Unset terminal flags
GetElecTerminalLoc(*thandle*, *index*, *array*)	Return electrical terminal location
SetElecTerminalLoc(*thandle*, *x*, *y*)	Assign electrical terminal location
ClearElecTerminalLoc(*thandle*, *x*, *y*)	Delete symbolic duplicate location
Physical Terminals	
ListPhysTerminals()	List physical cell contact terminals
FindPhysTerminal(*name*, *use_p*, *xp*, *yp*)	Find a physical cell connection terminal
CreatePhysTerminal(*thandle*, *x*, *y*, *layer*)	Create new linkage to layout terminal
HasPhysTerminal(*thandle*)	Check if terminal has physical component
DestroyPhysTerminal(*thandle*)	Remove and destroy layout terminal linkage
GetPhysTerminalLoc(*thandle*, *array*)	Return layout terminal location
SetPhysTerminalLoc(*thandle*, *x*, *y*)	Assign layout terminal location
GetPhysTerminalLayer(*thandle*)	Return associated layer name
SetPhysTerminalLayer(*thandle*, *layer*)	Set layer name for hinting
GetPhysTerminalGroup(*thandle*)	Return associated physical group number
GetPhysTerminalObject(*thandle*)	Return handle to associated object
Physical Conductor Groups	
Group()	Run extraction
GetNumberGroups()	Return number of groups
GetGroupBB(*group*, *array*)	Return bounding box of group
GetGroupNode(*group*)	Return node of group

`GetGroupName(`*group*`)`	Return net or formal terminal name
`GetGroupNetName(`*group*`)`	Return net name
`GetGroupCapacitance(`*group*`)`	Return group capacitance
`CountGroupObjects(`*group*`)`	Count physical objects in group
`ListGroupObjects(`*group*`)`	Return list of objects in group
`CountGroupVias(`*group*`)`	Count standard vias or via cells used in the group
`ListGroupVias(`*group*`)`	Return list of standard via or via cell instances used in the group
`CountGroupDevContacts(`*group*`)`	Count device contacts in group
`ListGroupDevContacts(`*group*`)`	Return list of device contacts in group
`CountGroupSubcContacts(`*group*`)`	Count subcircuit contacts in group
`ListGroupSubcContacts(`*group*`)`	Return list of subcircuit contacts in group
`CountGroupTerminals(`*group*`)`	Count cell connection terminals in group
`ListGroupTerminals(`*group*`)`	Return list of cell connection terminals in group
`ListGroupTerminalNames(`*group*`)`	Return list of cell contact terminal names in group
`CountGroupPhysTerminals(`*group*`)`	Count physical terminals in group
`ListGroupPhysTerminals(`*group*`)`	Return list of physical terminals in group
Physical Devices	
`ListPhysDevs(`*name*`, `*pref*`, `*indices*`, `*area_array*`)`	Return list of physical devices
`GetPdevName(`*device_handle*`)`	Return device name
`GetPdevIndex(`*device_handle*`)`	Return device index
`GetPdevDual(`*device_handle*`)`	Return corresponding electrical device
`GetPdevBB(`*device_handle*`, `*array*`)`	Return device bounding box
`GetPdevMeasure(`*device_handle*`, `*mname*`)`	Return device measurement
`ListPdevMeasures(`*device_handle*`)`	Return list of measurement keywords
`ListPdevContacts(`*device_handle*`)`	Return list of device contacts
`GetPdevContactName(`*dev_contact_handle*`)`	Return device contact name
`GetPdevContactBB(`*dev_contact_handle*`, `*array*`)`	Return device contact bounding box
`GetPdevContactGroup(`*dev_contact_handle*`)`	Return device contact conductor group
`GetPdevContactLayer(`*dev_contact_handle*`)`	Return device contact layer
`GetPdevContactDev(`*dev_contact_handle*`)`	Return device containing contact
`GetPdevContactDevName(`*dev_contact_handle*`)`	Return name of device containing contact
`GetPdevContactDevIndex(`*dev_contact_handle*`)`	Return index of device containing contact
Physical Subcircuits	
`ListPhysSubckts(`*name*`, `*index*`, `*l*`, `*b*`, `*r*`, `*t*`)`	Return list of physical subcircuits
`GetPscName(`*subckt_handle*`)`	Return master name of physical subcircuit
`GetPscIndex(`*subckt_handle*`)`	Return index of physical subcircuit
`GetPscIdNum(`*subckt_handle*`)`	Return id number of physical subcircuit
`GetPscInstName(`*subckt_handle*`)`	Return instance name of physical subcircuit
`GetPscDual(`*subckt_handle*`)`	Return corresponding electrical subcircuit
`GetPscBB(`*subckt_handle*`, `*array*`)`	Return physical subcircuit bounding box
`GetPscLoc(`*subckt_handle*`, `*array*`)`	Return physical subcircuit placement location
`GetPscTransform(`*subckt_handle*`, `*type*`, `*array*`)`	Return physical subcircuit orienttion string
`ListPscContacts(`*subckt_handle*`)`	Return list of contacts
`IsPscContactIgnorable(`*subc_contact_handle*`)`	Return 1 if contact to ignored subcircuit

`GetPscContactName(`*subc_contact_handle*`)`	Return name of subcircuit
`GetPscContactGroup(`*subc_contact_handle*`)`	Return conductor group of contact
`GetPscContactSubcGroup(`*subc_contact_handle*`)`	Return group of contact in subcircuit
`GetPscContactSubc(`*subc_contact_handle*`)`	Return subcircuit containing contact
`GetPscContactSubcName(`*subc_contact_handle*`)`	Return name of subcircuit containing contact
`GetPscContactSubcIndex(`*subc_contact_handle*`)`	Return index of subcircuit containing contact
`GetPscContactSubcIdNum(`*subc_contact_handle*`)`	Return id number of subcircuit containing contact
`GetPscContactSubcInstName(`*subc_contact_handle*`)`	Return instnce name of subcircuit containing contact
Electrical Devices	
`ListElecDevs(`*regex*`)`	Return list of electrical devices
`SetEdevProperty(`*devname*`, `*prpty*`, `*string*`)`	Set electrical device property
`GetEdevProperty(`*devname*`, `*prpty*`)`	Return electrical device property
`GetEdevObj(`*devname*`)`	Return electrical device subcell object
Resistance/Inductance Extraction	
`ExtractRL(`*conductor_zoidlist*`, `*layername*`, `*r_or_l*`, `*array*`, `*term*`, ...)`	Extract resistance or inductance from object
`ExtractNetResistance(`*net_handle*`, `*spicefile*`, `*array*`, `*term*`, ...)`	Extract resistance from wire net

Functions for electrical schematic editing:

Schematic Editor Functions	
Output Generation	
`Connect(`*for_spice*`)`	Internally process the schematic
`ToSpice(`*spicefile*`)`	Write SPICE file
Electrical Nodes	
`IncludeNoPhys(`*flag*`)`	Set nophys property usage
`GetNumberNodes()`	Return number of nodes in circuit
`SetNodeName(`*node*`, `*name*`)`	Set text name for node
`GetNodeName(`*node*`)`	Return text name for node
`GetNodeNumber(`*name*`)`	Return node number for named node
`GetNodeGroup(`*node*`)`	Return corresponding group for node
`ListNodePins(`*node*`)`	Return list of connected cell contact terminals
`ListNodeContacts(`*node*`)`	Return list of connected instance terminals
`GetNodeContactInstance(`*terminal_handle*`)`	Return handle to instance providing contact
`ListNodePinNames(`*node*`)`	Return list of connected cell contact terminal names
`ListNodeContactNames(`*node*`)`	Return list of connected instance terminal names
Symbolic Mode	
`IsShowSymbolic()`	True if current cell displayed symbolically in main window
`ShowSymbolic(`*show*`)`	Turn on/off symbolic display
`SetSymbolicFast(`*symb*`)`	Set symbolic mode of current cell, no display update
`MakeSymbolic()`	Create simple symbolic representation

F.1 Main Functions 1

F.1.1 Current Cell

(int) `Edit(`*name*`,` *symname*`)`

This function will read in the named file or cell and make it, or one of the cells in the hierarchy, the current cell. If the present cell has been modified, in graphics mode the user is prompted for whether to save the cell before reading the new one. The *name* argument can be null or empty, in which case the user will be prompted for a file or cell to open for editing, if in graphics mode. If not in graphics mode, an empty cell is created in memory and made the current cell.

The *name* provided can be an archive file, the name of an *Xic* cell, a library file, or the "database name" of a Cell Hierarchy Digest (CHD). If a CHD name or the name of an archive file is given, the name of the cell to open can be provided as *symname*. If *symname* is null or empty, The CHD's default cell, or the top level cell (the one not used as a subcell by any other cells in the file) is the one opened for editing. If there is more than one top level cell, in graphics mode the user is presented with a pop-up choice menu and asked to make a selection. If the file is a library file, the *symname* can be given, and it should be one of the reference names from the library, or the name of a cell defined in the library. If *symname* is null or empty, in graphics mode a pop-up listing the library contents will appear, allowing the user to select a reference or cell. If not in graphics mode, and the cell to edit can not be determined, the current cell is unchanged, and nothing is read.

See the table in 14.1 for the features that apply during a call to this function. This function is consistent with the **Open** menu command in that cell name aliasing, layer filtering and modification, and scaling are not available (unlike in the pre-3.0.0 version of this function). If these features are needed, the vt OpenCell function should be used instead.

The return value is one of the following integers, representing the command status:

-2	The function call was reentered. This is not likely to happen in scripts.
-1	The user aborted the operation.
0	The open failed: bad file name, parse error, etc.
1	The operation succeeded.
2	The read was successful on an archive with multiple top-level cells but the cells to edit can't be determined. The current cell has not been set, but the cells are in memory. The second argument could have been used to resolve the ambiguity.
3	The cell name was the name of the device library or model library file, which has been opened for text editing (in graphic mode only).

(int) `OpenCell(`*name*`,` *symname*`,` *curcell*`)`

This function will read a file into memory, similar to the `Edit` function. The first two arguments are the same as would be passed to `Edit`. The third argument is a boolean value.

See the table in 14.1 for the features that apply during a call to this function.

If *curcell* is nonzero, then this function will behave like the `Edit` function in switching the current cell to a newly-read cell. The only difference from `Edit` is that scaling, layer filtering and aliasing, and cell name modification are allowed, as in the pre-3.0.0 versions of the `Edit` function. The return values are those listed for the `Edit` function.

If *curcell* is zero, the new cell will not be the current cell. Once in memory, the cell is available by its simple name, for use by the `Place` function for example. If *name* is the name of an archive or library file, *symname* is the cell or reference to open, similar to the `Edit` function. In this mode, the return value is 1 on success, 0 otherwise.

(int) `TouchCell(`*cellname*`,` *curcell*`)`
If no cell exists in the current symbol table for the current mode with the given name, create an empty cell for *cellname* and add it to the symbol table. If the boolean *curcell* is true, switch the current cell to *cellname*. This can be much faster than `Edit` or `OpenCell` for cells already in memory. The return value is -1 on error, 0 if no new cell was created, or 1 if a new cell was created.

(int) `RegisterSubMasters(`*archive*`)`
Suppose that one has a collection of pcell sub-master *Xic* cells that have been imported from a foreign OpenAccess tool such as Virtuoso. These are assumed to not be portable pcells. One would like to use these cells to resolve pcells when reading directly from the OpenAccess database. There are two issues: 1) the system needs to know that these cells are available, and 2) one has to remap the cell names. The first issue is fixed simply by making the sub-masters available through the library mechanism. The second issue is due to the simple naming convention of the sub-master instantiations, which suffixes the pcell name with "`$$`" followed by an integer. The integer is a count of when the cell was generated, and is consistent with the design output at the time, but there is no guarantee the the names are consistent with the design at other times.

This function will read a collection of cells into a temporary symbol table. Those that are pcell sub-masters have the property strings entered into the internal pcell database, under the existing cell name. This will cause the correct cell name to be associated with a given parameter set. The cells are not saved, but the entries in the pcell table persist so that resolution, when reading OpenAccess or otherwise, will reference the correct cells. The cell collection must be available through an open library, and this function must be run before loading the design.

The argument is either a path to a directory containing native pcell sub-master cells, or a path to an archive file that contains the cells. The return is 1 on success, 0 otherwise with a message available from `GetError`; This functionality is also available with the **!preload** command.

(int) `Push(`*object_handle*`)`
This function will push the editing context to the cell of the instance referenced by the handle, that is, make it the currrent cell. The handle is the return value from the `SelectHandle` or `AreaHandle` functions. This is similar to the **Push** command in *Xic*. The editing context can be restored with the `Pop` function. If the instance is an array, the 0,0 element will be pushed (see `PushElement`).

If successful, 1 is returned, otherwise 0 is returned. This function will fail if the handle passed is not a handle to an object list.

This function implicitly calls `Commit` before the context change.

(int) `PushElement(`*object_handle*`,` *xind*`,` *yind*`)`
This is very similar to `Push`, but allows passing indices which select the instance element to push if the instance is arrayed. The indices are always effectively 0 in the `Push` function. An out of range index value will cause the function to return 0 and not push the context. If both index values are zero, the function is identical to `Push`. The selection of the array element only affects the graphical display.

This function implicitly calls `Commit` before the context change.

(int) `Pop()`
This function will pop the editing context to the parent cell, to be used after the `Push` function or a **Push** command in *Xic*. The `Pop` function always returns 1, and has no effect if there was no corresponding push.

This function implicitly calls `Commit` before the context change.

(string) `NewCellName()`
This function returns a string which is a valid cell name that does not conflict with any cell in the

current symbol table. The cell is not actually created. This can be used with the `Edit` function to open a new cell for editing, similar to the **New** button in the **File Menu**. This function never fails.

(string) `CurCellName()`
The return value of this function is a string containing the name of the current cell.

(string) `TopCellName()`
The return value of this function is a string containing the name of the top level cell in the hierarchy being edited. This is different from the current cell name while in a subedit (i.e., the **Push** command is active).

(string) `FileName()`
This function returns the name of the file from which the current cell was read. If there is no such file, a null string is returned.

(int) `CurCellBB(`*array*`)`
This function will return the bounding box of the current cell, in microns, in the *array*, as l, b, r, t. The array must have size 4 or larger. The function returns 1 on success, 0 if there is no current cell.

In electrical mode, the bounding box returned will be for the schematiic or symbolic representation, matching how the cell is displayed in the main window. See the `CellBB` function for an alternative.

(int) `SetCellFlag(`*cellname*`, `*flagname*`, `*set*`)`
This will set a flag (see 9.4.3) in the cell whose name is passed as the first argument. If this argument is 0, or a null or empty string, the current cell is understood. The second argument is a string giving the flag name. This must be the name of a user-modifiable flag. The third argument is a boolean indicating the new flag state, a nonzero value will set the flag, zero will unset it. The return value is the previous flag status (0 or 1), or -1 on error. On error, a message can be obtained from `GetError`.

Warning: This affects the user flags directly, and does **not** update the property used to hold flag status that is written to disk when the cell is saved. These flags should be set by setting the **Flags** property (property number 7105) with `AddProperty` or *AddCellProperty*, if the values need to persist when the cell is written to disk and reread.

(int) `GetCellFlag(`*cellname*`, `*flagname*`)`
This will query a flag (see 9.4.3) in the cell whose name is passed as the first argument. If this argument is 0, or a null or empty string, the current cell is understood. The second argument is a string giving the flag name, which can be any or the flag names. The return value is the flag status (0 or 1), or -1 on error. On error, a message can be obtained from `GetError`.

(int) `Save(`*newname*`)`
This function will save to disk file the current cell, and its descendents if the cell originated from an archive file. If the argument is null or the empty string, the current cell name is used, suffixed with one of the following if saving as an archive:

CGX	`.cgx`
CIF	`.cif`
GDSII	`.gds`
OASIS	`.oas`

The default format will be the format of the original input file, though format conversion can be imposed by adding one of these suffixes or "`.xic`" to *newname*. The cell is saved unconditionally; there is no user prompt.

See the table in 18.10 for the features that apply during a call to this function.

This function returns 1 on success, 0 otherwise. On error, a message is likely available from `GetError`.

(int) `UpdateNative(`*dir*`)`

This will write to disk all of the modified cells in the current hierarchy as native cell files in the directory given as the argument. If the argument is null or empty, cells will be written in the current directory. The return value is the number of cells written.

Note that only modified or internally created cells will be written. To write all cells as native cell files, use the `ToXIC` function.

F.1.2 Cell Info

(int) `CellBB(`*cellname*`,` *array* `[,` *symbolic*`])`

This function will return the bounding box of the named cell in the current mode, in microns, in the array, as l, b, r, t. If *cellname* is null or empty, the current cell is used. The array must have size 4 or larger. The function returns 1 on success, 0 if the cell is not found in memory.

The optional boolean third argument applies to electrical cells. If not given or set to false, the schematic bounding box is always returned. If this argument is true, and the cell has a symbolic representation, the symbolic representation bounding box is returned, or the function fails and returns 0 if the cell has no symbolic representation.

(stringlist_handle) `ListSubcells(`*cellname*`,` *depth*`,` *array*`,` *incl_top*`)`

This function returns a handle to a sorted list of subcell master names found under the named cell, to the given depth, and only if instantiated so as to overlap a rectangular area (if given). These apply to the current mode, electrical or physical. If *cellname* is null or empty, the current cell is used. The *depth* is the search depth, which can be an integer which sets the maximum depth to search (0 means search *cellname* only and return its subcell names, 1 means search *cellname* plus its subcells, etc., and a negative integer sets the depth to search the entire hierarchy). This argument can also be a string starting with '`a`' such as "`a`" or "`all`" which indicates to search the entire hierarchy.

The cell will be read into memory if not already there. The function fails if the cell can not be found.

The *array* argument can be passed 0, which indicates no area testing. Otherwise, the array should be size four or larger, with the values being the left (*array*[0]), bottom, right, and top coordinates of a rectangular region of *cellname*. Only cells that are instantiated such that the instance bounding box, when reflected to top-level coordinates, intersects the region will be listed.

If the boolean *incl_top* is nonzero, the top cell name (*cellname*) will be included in the list, unless an array is given and there is no overlap with the top cell.

The return is a handle to a list of cell names, and can be empty. The `GenCells` or `ListNext` functions can be used to iterate through the list.

(stringlist_handle) `ListParents(`*cellname*`)`

This function returns a list of cell names, each of which contain an instance of the cell name passed as the argument. These apply to the current mode, electrical or physical. If *cellname* is null or empty, the current cell is used.

The function fails if the cell can not be found in memory.

The return is a handle to a list of cell names, and can be empty. The `GenCells` or `ListNext` functions can be used to iterate through the list.

(stringlist_handle) `InitGen()`

This function returns a handle to a list of names of cells used in the hierarchy of the current cell, either the physical or electrical part according to the current mode. Each cell is listed once only, and all cells are listed, including the current cell which is returned last.

The return is a handle to a list of cell names, and can be empty. The `GenCells` or `ListNext` functions can be used to iterate through the list.

(stringlist_handle) `CellsHandle(`*cellname*`,` *depth*`)`

This function returns a handle to a list of subcell names found in *cellname*, to the given hierarchy depth. If *cellname* is null or empty, the current cell is used. The *depth* is the search depth, which can be an integer which sets the maximum depth to search (0 means search *cellname* only and return its subcell names, 1 means search *cellname* plus its subcells, etc., and a negative integer sets the depth to search the entire hierarchy). This argument can also be a string starting with 'a' such as `"a"` or `"all"` which indicates to search the entire hierarchy. The listing order is as a tree, with a subcell listed followed by the descent into that subcell.

The cell will be read into memory if not already there. The function fails if the cell can not be found.

With "`all`" passed, the output is similar to that of the `InitGen` function, except that the top-level cell name is not listed, and duplicate entries are not removed (`ListUnique` can be called to remove duplicate names).

Be aware that the listing will generally contain lots of duplicate names. This function is not recommended for general hierarchy traversal.

The return is a handle to a list of cell names, and can be empty. The `GenCells` or `ListNext` functions can be used to iterate through the list.

(string) `GenCells(`*stringlist_handle*`)`

This function returns a string containing the name of one of the elements in the list whose handle is passed as the argument. It advances the handle to point to the next name. The argument can be the return value from one of the functions above, or any *stringlist_handle* variable. A different name is returned for each call. The null string is returned after all names have been returned. This is identical to the `ListNext` function.

Example:

This script will list all of the cells in the current hierarchy:

```
i = InitGen()
while ((name = GenCells(i)) != 0)
   Print(name)
end
```

F.1.3 Database

`Clear(`*cellname*`)`

If *cellname* is not empty, any matching cell and all its descendents are cleared from the database, unless they are referenced by another cell not being cleared. If *cellname* is null or empty, the entire database is cleared. This function is obviously very dangerous.

`ClearAll(`*clear_tech*`)`

This will clear all cells from the present symbol table, clear and delete any other symbol tables that may be defined, and revert the layer database. If the boolean argument is nonzero, layers read from the technology file will be cleared, otherwise the layer database is reverted to the state just after the technology file was read. This function does **not** automatically open a new cell. This is for server mode, to give the system a good scrubbing between jobs.

(int) `IsCellInMem(`*cellname*`)`
This function returns 1 if the string *cellname* is the name of a cell in the current symbol table, 0 otherwise. If the string contains a path prefix, it will be ignored, and the last (filename) component used for the test.

(int) `IsFileInMem(`*filename*`)`
This will compare the string *filename* to the source file names saved with top-level cells in the current symbol table. If *filename* is a full path, the function returns 1 if an exact match is found. If *filename* is not rooted, the function returns 1 if the last path component matches. In either case, 0 is returned if no match is seen.

(int) `NumCellsInMem()`
This function returns an integer giving the number of cells in the current symbol table.

(stringlist_handle) `ListCellsInMem(`*options_str*`)`
This function returns a handle to a list of strings, sorted alphabetically, giving the names of cells found in the current symbol table.

A fairly extensive filtering capability is available, which is configured through a string passed as the argument. If 0 is passed, or the options string is null or empty, all cells will be listed.

The string consists of a space-separated list of keywords, each of which represents a condition for filtering. The cells listed will be the logical AND of all option clauses. The keysords are described with the **Cell List Filter** panel in 9.4.2.

(stringlist_handle) `ListTopCellsInMem()`
This function returns a handle to a list of strings, sorted alphabetically, giving the names of top-level cells in the current symbol table. These are the cells that are not used as subcells, in either physical or electrical mode.

(stringlist_handle) `ListModCellsInMem()`
This function returns a handle to a list of strings, sorted alphabetically, giving the names of modified cells in the current symbol table. A cell is modified if the contents have changed since the cell was read or last written to disk.

(stringlist_handle) `ListTopFilesInMem()`
This function returns a handle to a list of strings, alphabetically sorted, giving the source file names of the top-level cells in the current symbol table.

F.1.4 Symbol Tables

(string) `SetSymbolTable(`*tabname*`)`
This function will set the current symbol table to the table named in the argument string. If the *tabname* is null or empty, the default "`main`" table is understood. If a table by the given name does not exist, a new table will be created for that name.

The return value is a string giving the name of the active table before the switch.

(int) `ClearSymbolTable(`*destroy*`)`
This function will clear or destroy the current symbol table. If the boolean argument is nonzero, and the current table is not the "`main`" table, the current table and its contents will be destroyed.

Otherwise, the current table will be cleared, i.e., all contained cells will be destroyed. If the current symbol table is destroyed, a new current table will be installed from among the internal list of existing tables.

This function always returns 1.

(string) `CurSymbolTable()`
This function returns a string giving the name of the current symbol table.

F.1.5 Display

(int) `Window(`*x*`,` *y*`,` *width*`,` *win*`)`
The window view is changed so that it is centered at x, y and has width set by the third argument. If the *width* argument is less than or equal to zero, a centered, full view of the current cell is obtained. In this case, the x, y arguments are ignored. The *win* is an integer 0–4 which specifies the window:

0	Main drawing window
1–4	Sub-window (number as shown in title bar)

The function returns 1 on success, 0 if the indicated window does not exist.

(int) `GetWindow()`
This function returns the window number of the drawing window that contains the pointer. The window number is an integer 0–4:

0	Main drawing window
1–4	Sub-window (number as shown in title bar)

If the pointer is not in a drawing window, 0 is returned.

(int) `GetWindowView(`*win*`,` *array*`)`
This function returns the view area (visible cell coordinates) of the given window *win*, which is an integer 0–4 where 0 is the main window and 1–4 represent sub-windows. The view coordinates, in microns, are returned in the *array*, in order L, B, R, T. On success, 1 is returned, otherwise 0 is returned and the *array* is untouched.

(int) `GetWindowMode(`*win*`)`
This function returns the display mode of the given window *win*, which is 0 for physical mode, 1 for electrical, or -1 if the window does not exist. The argument is an integer 0–4, where 0 represents the main window and 1–4 indicate sub-windows. This function is identical to `CurMode`.

(int) `Expand(`*win*`,` *string*`)`
This sets the expansion mode for the display in the window specified in *win*. The *win* argument is an integer 0–4, where 0 refers to the main window, and 1–4 correspond to the sub-windows brought up with the **Viewport** command. The *string* contains characters which modify the display mode, as would be given to the **Expand** command in the **View Menu**.

integer	set expand level
`n`	set level to 0
`a`	expand all
+	increment expand level
−	decrement expand level

(int) `Display(`*display_string*`,` *win_id*`,` *l*`,` *b*`,` *r*`,` *t*`)`
This function will render the current cell in a foreign X window. The X window id is passed as an integer in the second argument. The first argument is the X display string corresponding to the

server in which the window is cached. The remaining arguments set the area to be displayed, in microns. The function returns 1 upon success, 0 otherwise. This function is useful for rendering a layout if interactive graphics is not enabled, such as in server mode. This function will not work under Microsoft Windows.

This is a primitive to allow *Xic* to export graphics rendering capability. The intention is that this might be used in a Tk script (for example) that is otherwise using *Xic* in server mode as a back-end. The machine containing the window to be drawn into must allow X access to the machine running the *Xic* server (see the xhost Unix command).

One can demonstrate the capability as follows. The "`xwininfo -children`" Unix command can be used to find the window id of a suitable *child* window in a running application. The top-level window given from xwininfo without the "`-children`" argument is generally obscured by child windows, so this won't work. For example, an xterm window has a single child, which is the id to use. In server mode, a cell must be loaded for editing with the `Edit` function. Then, a `Display` command can be given, something like

```
Display(":0", 0x1800015, -100, -100, 100, 100)
```

The `":0"` is the display name for the local machine, assuming that the *Xic* server is also running on this machine. In general, this is the same as the DISPLAY environment variable, in the form *hostname*`:0`. The second argument is the window id returned from xwininfo. The remaining arguments set the area to display. After giving the command, the window should be overwritten with a display similar to a drawing window in *Xic*. However, if the window is redrawn, it will revert to its previous contents. The user must set up expose event handling in a real application. The suggested way to do this is to pass the id of a pixmap to *Xic*, and then copy the pixmap to the destination window. This is usually faster than a direct write, and the pixmap can be used for backing store for expose events.

(int) `FreezeDisplay(`*freeze*`)`
When this function is called with a nonzero argument, the graphical display in the drawing windows will be frozen until a subsequent call of this function with a zero argument, or the script terminates. This is useful for speeding execution, and eliminating distracting screen drawing while a script is running. When the function is called with a zero argument, all drawing windows are refreshed.

(int) `Redraw(`*win*`)`
This function will redraw the window indicated by the argument, which is 0 for the main window or 1–4 for the sub-windows. The function returns 0 if the argument does not correspond to an existing window, 1 otherwise.

F.1.6 Exit

`Exit()`
Calling this function terminates execution of the script.

`Halt()`
Calling this function terminates execution of the script, equivalent to `Exit`.

F.1.7 Annotation

(int) `AddMark(`*type*`,` *arguments* `...)`
This function will add a "user mark" to a display list, which is rendered as highlighting in the

current cell. These can be used for illustrative purposes. The marks are not included in the design database, but are persistent to the current cell and are remembered as long as the current cell exists in memory. Any call can have associated marks, whether electrical or physical. Marks are shown in any window displaying the cell as the top level. Marks are not shown in expanded subcells.

The arguments that follow the type argument vary depending upon the type. The type argument can be an integer code, or a string whose first character signifies the type. The return value, if nonzero, is a unique mark id, which can be passed to `EraseMark` to erase the mark. A zero return indicates that an error occurred.

The table below describes the marks available. All coordinates and dimensions are in microns, in the coordinate system of the current cell. Each mark takes an optional attribute argument, which is an integer whose set bits indicate a display property. These bits are

bit 0: Draw with a textured (dashed) line if set, otherwise use a solid line.

bit 1: Cause the mark to blink, using the selection colors.

bit 2: Render the mark in an alternate color (bit 1 is ignored).

Type: 1 or `"l"`
Arguments: *x1*, *y1*, *x2*, *y2* [, *attribute*]
Draw a line segment from *x1*,*y1* to *x2*,*y2*.

Type: 2 or `"b"`
Arguments: *l*, *b*, *r*, *t* [, *attribute*]
Draw an open box, *l*,*b* is lower-left corner and *r*,*t* is upper-right corner.

Type: 3 or `"u"`
Arguments: *xl*, *xr*, *yb* [, *yt*, *attribute*]
Draw an open triangle. The two base vertices are *xl*,*yb* and *xr*,*yb*. The third vertex is (*xl*+*xr*)/2,*yt*. If *yt* is not given, it is set to make the triangle equilateral.

Type: 4 or `"t"`
Arguments: *yl*, *yu*, *xb* [, *xt*, *attribute*]
Draw an open triangle. The two lower vertices are *xb*,*yl* and *xb*,*yu*. The third vertex is *xt*,(*yl*+*yu*)/2. If *xt* is not given, it is set to make the triangle equilateral.

Type: 5 or `"c"`
Arguments: *xc*, *yc*, *rad* [, *attribute*]
Draw a circle of radius *rad* centered at *xc*,*yc*.

Type: 6 or `"e"`
Arguments: *xc*, *yc*, *rx*, *ry* [, *attribute*]
Draw an ellipse centered at xc,yc using radii rx and ry.

Type: 7 or `"p"`
Arguments: *numverts*, *xy_array* [, *attribute*]
Draw an open polygon or path. The number of vertices is given first, followed by an array of size `2*`*numverts* or larger that contains the vertex coordinates as x-y pairs. For a polygon, The vertex list should be closed, i.e., the first and last vertices listed (and counted) should be the same.

Type: 8 or `"s"`
Arguments: *string*, *x*, *y* [, *width*, *height*, *xform*, *attribute*]
Draw a text string. The string is followed by the coordinates of the reference point, which for default justification is the lower-left corner of the bounding box. The *width*, *height*, and *xform* arguments are analogous to those of the `Label` script function, providing the rendering size and justification and transformation information. Unlike the `Label` function, the settings of

the `Justify` and `UseTransform` functions are ignored, transformation and justification must be set through the *xform* argument.

(int) `EraseMark(`*id*`)`
Remove a mark from the "user marks" display list. The argument is the id number returned from `AddMark`. If zero is passed instead, all marks will be erased. The return value is 1 if any marks were erased.

(int) `DumpMarks(`*filename*`)`
This function will save the marks currently defined in the current cell to a file. If the argument is null or empty (or scalar 0), a file name will be composed: *cellname*.*mode*.`marks`, where *mode* is "`phys`" or "`elec`". The return is the number of marks written, or -1 if error. On error, a message may be available from `GetError`. If 0, no file was produced, as no marks were found.

(int) `ReadMarks(`*filename*`)`
This function will read the marks found in a file into the current cell. The file must be in the format produced by `DumpMarks`, and apply to the same name and display mode as the current cell. A null or empty or 0 argument will imply a cell name composed as described for `DumpMarks` The return value is the number of marks read, or -1 if error. On error, a message may be available from `GetEreror`.

F.1.8 Ghost Rendering

The `PushGhost/PopGhost` functions are useful in scripts where an object is created, and the user must click to place the object. The object's outline can be drawn and attached to the pointer, facilitating placement. Example:

```
array[2000]
# create some shape in array, nverts is actual size
...
ShowPrompt("Click to locate new object");
xy[2]
PushGhost(array, nverts)
ShowGhost(8)
if !Point(xy)
    Exit()
end
ShowGhost(0)
PopGhost()
# use xy to create object in database
```

(int) `PushGhost(`*array*`, `*numpts*`)`
This function allows a polygon to be added to the list of polygons used for dynamic highlighting with the `ShowGhost` function. The outline of the polygon will be "attached" to the mouse pointer. The return value is the number of polygons in the list, after the present one is added. The *array* is an array of x-y values forming the polygon. The *numpts* value is the number of x-y pairs that constitute the polygon. If this value is less than 2 or greater than the real size of the array, the real size of the array will be assumed. The second argument is useful when the polygon data do not entirely fill the array, and can be set to 0 otherwise.

(int) `PushGhostBox(`*left*`,` *bottom*`,` *right*`,` *top*`)`
This function is similar to `PushGhost`. It allows a box outline to be added to the list of polygons used for ghosting with the `ShowGhost` function. The outline of the box will be "attached" to the mouse pointer. The return value is the number of polygons in the list, after the present one is added. The arguments are the coordinates of the lower left and upper right corners of the box, where "0" is the point attached to the mouse pointer. The `PopGhost` function is used to remove the most recently added object from the list.

(int) `PushGhostH(`*object_handle*`,` *all*`)`
Push the outline of the figure referenced by the handle onto the ghost list. If boolean *all* is true, push all objects in the list represented by the handle, otherwise push the single object at the head of the list. The return value is an integer count of the number of outlines added to the ghost list.

(int) `PopGhost()`
This function removes the last ghosting polygon passed to `PushGhost` or `PushGhostBox` from the internal list, and returns the number of polygons remaining in the list.

(int) `ShowGhost(`*type*`)`
Show dynamic highlighting. This function turns on/off the ghosting, i.e., the display of certain features which are "attached" to the mouse pointer. The argument is one of the numeric codes from the table below.

0	Turn off ghosting
1	full-screen horiz line, snapped to grid
2	full-screen vert line, snapped to grid
3	full-screen horiz line, not snapped
4	full-screen vert line, not snapped
5	vector from last point location to pointer
6	box, snapped to grid
7	box, not snapped
8	display polygon list from `PushGhost`
9	vector from last point location to pointer
10	vector from last point location to pointer
11	vector from last point location to pointer

The modes 5, 9, 10, and 11 draw a vector from the last button 1 down location to the pointer. Mode 5 snaps to the grid, and snaps the angle to multiples of 45 degrees when the angle is close. If the `Constrain45` variable is set, the angle is strictly constrained to multiples of 45 degrees. Mode 9 is similar, but does not snap to grid. Mode 10 is similar, but there are no angle constraints, except that implicit in snapping to the grid. Mode 11 is similar, but there are no angle constraints and no grid snapping.

With the ghosting enabled, the `Point` function returns coordinates that are snapped to grid or not depending on the mode passed to `ShowGhost`. Modes 1, 2, 5, 6, 8, and 10 are snapped to grid.

If the `UseTransform` function has been called to enable use of the current transform, the current transform will be applied to the displayed objects when using mode 8. The translation supplied to `UseTransform` is ignored (the translation tracks the mouse pointer).

F.1.9 Graphics

The following functions represent an interface for exporting graphics to a "foreign" X window. In particular, the interface can be used to draw into a window owned by a `Tk` script. This interface is not available on Microsoft Windows.

(handle) `GRopen(`*display*`,` *window*`)`
This function returns a handle to a graphical interface that can be used to export graphics to a foreign X window, possibly on another machine. The first argument is the X display string, corresponding to the server which owns the target window. The second argument is the X window id of the target window to which graphics rendering is to be exported. If all goes well, and the user has permission to access the window, a positive integer handle is returned. If the open fails, 0 is returned. The handle should be closed with the `Close` function when done.

(int) `GRcheckError()`
This function returns 1 if the previous operation by any of the GR interface functions caused an X error, 0 otherwise.

(drawable) `GRcreatePixmap(`*handle*`,` *width*`,` *height*`)`
This function returns the X id of a new pixmap. The first argument is a handle returned from `GRopen`. The remaining arguments set the size of the pixmap. If the operation fails, 0 is returned.

(int) `GRdestroyPixmap(`*handle*`,` *pixmap*`)`
This function destroys a pixmap created with `GRcreatePixmap`. The first argument is a handle returned from `GRopen`. The second argument is the pixmap id returned from `GRcreatePixmap`. The function returns 1 on success, 0 if there was an error.

(int) `GRcopyDrawable(`*handle*`,` *dst*`,` *src*`,` *xs*`,` *ys*`,` *ws*`,` *hs*`,` *x*`,` *y*`)`
This function is used to copy area between drawables, which can be windows or pixmaps. The first argument is a handle returned from `GRopen`. The next two arguments are the ids of destination and source drawables. The area copied in the source drawable is given by the next four arguments. The coordinates are pixel values, with the origin in the upper left corner. If these four values are all zero, the entire source drawable is understood. The final two values give the upper left corner of the copied-to area in the destination drawable.

(int) `GRdraw(`*handle*`,` *l*`,` *b*`,` *r*`,` *t*`)`
This function renders an *Xic* cell. The first argument is a handle returned from `GRopen`. The remaining arguments are the coordinates of the cell to render, in microns. The action is the same as the `Display` function. The function returns 1 on success, 0 if there was an error.

(int) `GRgetDrawableSize(`*handle*`,` *drawable*`,` *array*`)`
This function returns the size, in pixels, of a drawable. The first argument is a handle returned from `GRopen`. The second argument is the id of a window or pixmap. The third argument is an array of size two or larger that will contain the pixel width and height of the drawable. Upon success, 1 is returned, and the array values are set, otherwise 0 is returned. The width is in the 0'th array element.

(drawable) `GRresetDrawable(`*handle*`,` *drawable*`)`
This function allows the target window of the graphical context to be changed. Then, the rendering functions will draw into the new window or pixmap, rather than the one passed to `GRopen`. The return value is the previous drawable id, or 0 if there is an error.

(int) `GRclear(`*handle*`)`
This function clears the window. The argument is a handle returned from `GRopen`. Upon success, 1 is returned, otherwise 0 is returned.

(int) `GRpixel(`*handle*`,` *x*`,` *y*`)`
This function draws a single pixel at the pixel coordinates given in the second and third arguments, using the current color. The first argument is a handle returned from `GRopen`. Upon success, 1 is returned, otherwise 0 is returned.

(int) `GRpixels(`*handle*`,` *array*`,` *num*`)`
This function will draw multiple pixels using the current color. The first argument is a handle returned from `GRopen`. The second argument is an array of pixel coordinates, taken as x-y pairs. The third argument is the number of pixels to draw (half the length of the array). Upon success, 1 is returned, otherwise 0 is returned.

(int) `GRline(`*handle*`,` *x1*`,` *y1*`,` *x2*`,` *y2*`)`
This function renders a line using the current color and line style. The first argument is a handle returned from `GRopen`. The next four arguments are the endpoints of the line in pixel coordinates. Upon success, 1 is returned, otherwise 0 is returned.

(int) `GRpolyLine(`*handle*`,` *array*`,` *num*`)`
This function renders a polyline in the current color and line style. The first argument is a handle returned from `GRopen`. The second argument is an array containing vertex coordinates in pixels as x-y pairs. The line will be continued to each successive vertex. The third argument is the number of vertices (half the length of the array). Upon success, 1 is returned, otherwise 0 is returned.

(int) `GRlines(`*handle*`,` *array*`,` *num*`)`
This function renders multiple distinct lines, each using the current color and line style. The first argument is a handle returned by `GRopen`. The second argument is an array of coordinates, in pixels, which if taken four at a time give the x-y endpoints of each line. The third argument is the number of lines in the array (one fourth the array length). Upon success, 1 is returned, otherwise 0 is returned.

(int) `GRbox(`*handle*`,` *l*`,` *b*`,` *r*`,` *t*`)`
This function renders a rectangular area in the current color with the current fill pattern. The first argument is a handle returned from `GRopen`. The remaining arguments provide the diagonal vertices of the rectangle, in pixels. Upon success, 1 is returned, otherwise 0 is returned.

(int) `GRboxes(`*handle*`,` *array*`,` *num*`)`
This function renders multiple rectangles, each using the current color and fill pattern. The first argument is a handle returned from `GRopen`. the second argument is an array of pixel coordinates which specify the boxes. Taken four at a time, the values are the upper-left corner (x-y), width, and height. The third argument is the number of boxes represented in the array (one fourth the array length). Upon success, 1 is returned, otherwise 0 is returned.

(int) `GRarc(`*handle*`,` *x0*`,` *y0*`,` *rx*`,` *ry*`,` *theta1*`,` *theta2*`)`
This function renders an arc, using the current color and line style. The first argument is a handle returned from `GRopen`. The next two arguments are the pixel coordinates of the center of the ellipse containing the arc. The remaining arguments are the x and y radii, and the starting and ending angles. The angles are in radians, relative to the three-o'clock position, counter-clockwise. Upon success, 1 is returned, otherwise 0 is returned.

(int) `GRpolygon(`*handle*`,` *array*`,` *num*`)`
This function renders a polygon, using the current color and fill pattern. The first argument is a handle returned from `GRopen`. The second argument is an array containing the vertices, as x-y pairs of pixel coordinates. The third argument is the number of vertices (half the length of the array). The polygon will be closed automatically if the first and last vertices do not coincide. Upon success, 1 is returned, otherwise 0 is returned.

(int) `GRtext(`*handle*`,` *text*`,` *x*`,` *y*`,` *flags*`)`
This function renders text in the current color. The first argument is a handle returned form `GRopen`. The second argument is the text string to render. The next two arguments give the anchor point in pixel coordinates. If there is no transformation, this will be the lower-left of the

bounding box of the rendered text. The *flags* argument specifies a label flags word as used in *Xic* (see C.2). Only the bits of the least significant byte are likely to be recognized.

Upon success, 1 is returned, otherwise 0 is returned.

(int) `GRtextExtent(`*handle*`,` *text*`,` *array*`)`
This function returns the width and height in pixels needed to render a text string. The first argument is a handle returned from `GRopen`. The second argument is the string to measure. If the string is null or empty, a "typical" single character width and height is returned, which can be simply multiplied for the fixed-pitch font in use. The third argument is an array of size two or larger which will receive the width (0'th index) and height. The function returns 1 on success, 0 otherwise.

(int) `GRdefineColor(`*handle*`,` *red*`,` *green*`,` *blue*`)`
This function will return a color code corresponding to the given color. The first argument is a handle returned from `GRopen`. The next three arguments are color component values, each in a range 0–255, giving the red, green, and blue intensity. The return value is a color code representing the nearest displayable color to that given. If an error occurs, 0 (black) is returned. The returned color code can be passed to `GRsetColor` to actually change the drawing color.

(int) `GRsetBackground(`*handle*`,` *pixel*`)`
This function sets the default background color assumed by the graphics context. The first argument is a handle returned from `GRopen`. The second argument is a color code returned from `GRdefineColor`. Upon success, 1 is returned, otherwise 0 is returned.

(int) `GRsetWindowBackground(`*handle*`,` *pixel*`)`
This function sets the color used to render the window background when the window is cleared. The first argument is a handle returned from `GRopen`. The second argument is a color code returned from `GRdefineColor`. The function returns 1 on success, 0 otherwise.

(int) `GRsetColor(`*handle*`,` *pixel*`)`
This function sets the current color, used for all rendering functions. The first argument is a handle returned from `GRopen`. The second argument is a color code returned from `GRdefineColor`. Upon success, 1 is returned, otherwise 0 is returned.

(int) `GRdefineLinestyle(`*handle*`,` *index*`,` *mask*`)`
This function defines a line style. The first argument is a handle returned from `GRopen`. The second argument is an index value 1–15 which corresponds to an internal line style register. The third argument is an integer value whose bits set the line on/off pattern. the pattern starts with the most significant '1' bit in the *mask*. The '1' bits will be drawn. The pattern continues to the least significant bit, and is repeated as the line is rendered. The indices 1–10 contain pre-defined line styles, which can be overwritten with this function. The `SetLinestyle` function is used to set the pattern actually used for rendering. Upon success, 1 is returned, otherwise 0 is returned.

(int) `GRsetLinestyle(`*handle*`,` *index*`)`
This function sets the line style used to render lines. The first argument is a handle returned from `GRopen`. The second argument is an integer 0–15 which corresponds to an internal style register. Index 0 is always solid, whereas the other values can be set with `GRdefineLinestyle`. The function returns 1 on success, 0 otherwise.

(int) `GRdefineFillpattern(`*handle*`,` *index*`,` *nx*`,` *ny* *array_string*`)`
This function is used to define a fill pattern for rendering boxes and polygons. The first argument is a handle returned from `GRopen`. The second argument is an integer 1–15 which corresponds to internal fill pattern registers. The next two arguments set the x and y size of the pixel map used for the fill pattern. These can take values of 8 or 16 only. The final argument is a character string

which contains the pixel map. The most significant bit of the first byte is the upper left corner of the map. The `SetFillpattern` function is used to set the fill pattern actually used for rendering. The function returns 1 on success, 0 otherwise.

(int) `GRsetFillpattern(`*handle*`,` *index*`)`
This function sets the fill pattern used for rendering boxes and polygons. The first argument is a handle returned from `GRopen`. The second argument is an integer index 0–15 which corresponds to internal fill pattern registers. The value 0 is always solid fill. The other values can be set with `GRdefineFillpattern`. Upon success, 1 is returned, otherwise 0 is returned.

(int) `GRupdate(`*handle*`)`
This function flushes the X queue and causes any pending operations to be performed. This should be called after completing a sequence of drawing functions, to force a screen update. Upon success, 1 is returned, otherwise 0 is returned.

(int) `GRsetMode(`*handle*`,` *mode*`)`
This function sets the drawing mode used for rendering. The first argument is a handle returned from `GRopen`. The second argument is one of the following:

0	normal drawing
1	XOR
2	OR
3	AND-inverted

Modes 2,3 are probably not useful on other than 8-plane displays. The function returns 1 on success, 0 otherwise.

F.1.10 Hard Copy

The following functions provide an interface for plot and graphical file output. This is completely outside of the normal printing interface.

(stringlist_handle) `HClistDrivers()`
This function returns a handle to a list of available printer drivers. The returned handle can be processed by any of the functions that operate on stringlist handles.

(int) `HCsetDriver(`*driver*`)`
This function will set the current print driver to the name passed (as a string). The name must be one of the internal driver names as returned from `HClistDrivers`. If the operation succeeds, the function returns 1, otherwise 0 is returned.

(string) `HCgetDriver()`
This function returns the internal name of the current driver. If no driver has been set, a null string is returned.

(int) `HCsetResol(`*resol*`)`
This function will set the resolution of the current driver to the value passed. The scalar argument should be one of the values supported by the driver, as returned from `HCgetResols`. If the resolution is set successfully, 1 is returned. If no driver has been set, or the driver does not support the given resolution, 0 is returned.

(int) `HCgetResol()`
This function returns the resolution set for the current driver, or 0 if no driver has been set or the driver does not provide settable resolutions.

(int) `HCgetResols(`*array*`)`
This function sets the array values to the resolutions supported by the current driver. The array must have size 8 or larger. The return value is the number of resolutions supported. If no driver has been set, or the driver has fixed resolution, 0 is returned.

(int) `HCsetBestFit(`*best_fit*`)`
This function will set or reset the "best fit" flag for the current driver. In best fit mode, the image will be rotated 90 degrees if this is a better match to the aspect ratio of the rendering area. If the operation succeeds, 1 is returned. If there is no driver set or the driver does not allow best fit mode, 0 is returned. If the argument is nonzero, best fit mode will be set if possible, otherwise the mode is unset.

(int) `HCgetBestFit()`
This function returns 1 if the current driver is in "best fit" mode, 0 otherwise.

(int) `HCsetLegend(`*legend*`)`
This function will set or reset the "legend" flag for the current driver. If set, a legend will be shown with the rendered image. If the operation succeeds, 1 is returned. If there is no driver set or the driver does not allow a legend, 0 is returned. If the argument is nonzero, the legend mode will be set if possible, otherwise the mode is unset.

(int) `HCgetLegend()`
This function returns 1 if the current driver has the "legend" mode set, 0 otherwise.

(int) `HCsetLandscape(`*landscape*`)`
This function will set or reset the "landscape" flag for the current driver. If set, the image will be rotated 90 degrees. If the operation succeeds, 1 is returned. If there is no driver set or the driver does not allow landscape mode, 0 is returned. If the argument is nonzero, the landscape mode will be set if possible, otherwise the mode is unset.

(int) `HCgetLandscape()`
This function returns 1 if the current driver has the "landscape" mode set, 0 otherwise.

(int) `HCsetMetric(`*metric*`)`
This function sets a flag in the current driver which indicates that the rendering area is given in millimeters. If not set, the values are taken in inches. This pertains to the values passed to the `HCsetSize` function. If the operation succeeds, 1 is returned. If there is no driver set, 0 is returned. If the argument is nonzero, the metric mode will be set if possible, otherwise the mode is unset.

(int) `HCgetMetric()`
This function returns 1 if the current driver has the "metric" mode set, 0 otherwise.

(int) `HCsetSize(`*x*`, `*y*`, `*w*`, `*h*`)`
This function sets the size and offset of the rendering area. The numbers correspond to the entries in the **Print Control Panel**. The values are scalars, in inches unless metric mode is in effect (with `HCsetMetric`) in which case the values are in millimeters. The values are clipped to the limits provided in the technology file. Most drivers accept 0 for one of w, h, indicating auto dimensioning mode. The function returns 1 on success, 0 if no driver has been set. Not all drivers use all four parameters, unused parameters are ignored.

(int) `HCgetSize(`*array*`)`
This function returns the rendering area parameters for the current driver. The array argument must have size 4 or larger. The values are returned in the order x, y, w, h. If the function succeeds, the values are set in the array and 1 is returned. Otherwise, 0 is returned.

(int) `HCshowAxes(`*style*`)`
This function sets the style or visibility of axes shown in plots of physical data (electrical plots never include axes). The argument is an integer 0–2, where 0 suppresses drawing of axes, 1 indicates plain axes, and 2 (or anything else) indicates axes with a box at the origin. The return value is the previous setting.

(int) `HCshowGrid(`*show*`, `*mode*`)`
This function determines whether or not the grid is shown in plots. If the first argument is nonzero, the grid will be shown, otherwise the grid will not be shown. The second argument indicates the type of data affected: zero for physical data, nonzero for electrical data. The return value is the previous setting.

(int) `HCsetGridInterval(`*spacing*`, `*mode*`)`
This function sets the grid spacing used in plots. The first argument is the interval in microns. The second argument indicates the type of data affected: zero for physical data, nonzero for electrical data. For electrical data, the spacing in microns is rather meaningless, except as being relative to the default which is 1.0. The return value is the previous setting.

(int) `HCsetGridStyle(`*linemod*`, `*mode*`)`
This function sets the line style used for the grid lines in plots. The first argument is an integer mask that defines the on-off pattern. The pattern starts at the most significant '1' bit and continues through the least significant bit, and repeats. Set bits are rendered as the visible part of the pattern. If the style is 0, a dot is shown at each grid point. Passing -1 will give continuous lines. The second argument indicates the type of data affected: zero for physical data, nonzero for electrical data. The return value is the previous setting.

(int) `HCsetGridCrossSize(`*xsize*`, `*mode*`)`
This applies only to grids with style 0 (dot grid). The *xsize* is an integer 0–6 which indicates the number of pixels to draw in the four compass directions around the central pixel. Thus, for nonzero values, the "dot" is rendered as a small cross. The second argument indicates the type of data affected: zero for physical data, nonzero for electrical data. The return value is 1 if the cross size was set, 0 if the grid style was nonzero in which case the cross size was not set.

(int) `HCsetGridOnTop(`*on_top*`, `*mode*`)`
This function sets whether the grid lines are drawn after the geometry ("on top") or before the geometry. If the first argument is nonzero, the grid will be rendered on top. The second argument indicates the type of data affected: zero for physical data, nonzero for electrical data. The return value is the previous setting.

(int) `HCdump(`*l*`, `*b*`, `*r*`, `*t*`, `*filename*`, `*command*`)`
This is the function which actually generates a plot or graphics file. The first four arguments set the area in microns in current cell coordinates to render. If these values are all 0, a full view of the current cell will be rendered. The next argument is the name of the file to use for the graphical output. If this string is null or empty, a temporary file will be used. Under Windows, the final argument is the name of a printer, as known to the operating system. These names can be obtained with `HClistPrinters`. Under Unix/Linux, the last argument is a command string that will be executed to generate a plot. In any case if this argument is null or empty, the plot file will be generated, but no further action will be taken. In the command string, the character sequence "`%s`" will be replaced by the file name. If the sequence does not appear, the file name will be appended. If successful, 1 is returned, otherwise 0 is returned, and an error message can be obtained with `HCerrorString`.

The *filename*, or the temporary file that is used if no *filename* is given, is *not* removed. The user must remove the file explicitly.

The Windows Native driver (Windows only) has slightly different behavior. For this driver, the command string must specify a printer name, and can not be null or empty. If *filename* is not null or empty, the output goes to that file and is *not* sent to the printer. Otherwise, the output goes to the printer.

(int) `HCerrorString()`
This function returns a string indicating the error generated by `HCdump`. If there were no errors, a null string is returned.

(stringlist_handle) `HClistPrinters()`
Under Microsoft Windows, this function returns a handle to a list of printer names available from the current host. The first name is the name of the default printer. The remaining names, alphabetized, follow. If there are no printers available, or if not running under Windows, the function returns 0. The returned names can be supplied to the `HCdump` function to initiate a print job.

(int) `HCmedia()`
This function sets the media index, which is used by the Windows Native driver under Microsoft Windows only. The media index sets the assumed paper size. The argument is one of the integers from the table below. The page dimensions are in points (1/72 inch).

Index	**Name**	**Width**	**Height**
0	Letter	612	792
1	Legal	612	1008
2	Tabloid	792	1224
3	Ledger	1224	792
4	10x14	720	1008
5	11x17	792	1224
6	12x18	864	1296
7	17x22 “C”	1224	1584
8	18x24	1296	1728
9	22x34 “D”	1584	2448
10	24x36	1728	2592
11	30x42	2160	3024
12	34x44 “E”	2448	3168
13	36x48	2592	3456
14	Statement	396	612
15	Executive	540	720
16	Folio	612	936
17	Quarto	610	780
18	A0	2384	3370
19	A1	1684	2384
20	A2	1190	1684
21	A3	842	1190
22	A4	595	842
23	A5	420	595
24	A6	298	420
25	B0	2835	4008
26	B1	2004	2835
27	B2	1417	2004
28	B3	1001	1417
29	B4	729	1032
30	B5	516	729

The returned value is the previous setting of the media index.

F.1.11 Keyboard

(int) `ReadMapfile(`*mapfile*`)`
Read and assert a keyboard mapping file, as generated from within *Xic* with the **Key Map** button in the **Attributes Menu**. If the *mapfile* is not rooted, it is searched for in the current directory, the user's home directory, and in the library search path, in that order. If success, 1 is returned, and the supplied mapping is installed. Otherwise, 0 is returned, and an error message is available from `GetError`.

F.1.12 Libraries

(int) `OpenLibrary(`*path_name*`)`
This function will open the named library. The name is either a full path to the library file, or the name of a library file to find in the search path. Zero is returned on error, nonzero on success.

(int) `CloseLibrary(`*path_name*`)`
This function will close the named library, or all user libraries if the argument is null. The *path_name* can be a full path to a previously opened library file, or just the file name. This function always returns 1.

F.1.13 OpenAccess

These functions provide an interface to the OpenAccess database. An OpenAccess exception triggered by these functions will generate a fatal error, terminating the script. The functions that return an integer that is not an explicit boolean result always return 1.

(string) `OaVersion()`
Return the version string of the connected OpenAccess database. If none, a null string is returned.

(int) `OaIsLibrary(`*libname*`)`
Return 1 if the library named in the string argument is known to OpenAccess, 0 if not.

(stringlist_handle) `OaListLibraries()`
Return a handle to a list of library names known to OpenAccess.

(stringlist_handle) `OaListLibCells(`*libname*`)`
Return a list of the names of cells contained in the OpenAccess library named in the argument.

(stringlist_handle) `OaListCellViews(`*libname*`,` *cellname*`)`
Return a handle to a list of view names found for the given cell in the given OpenAccess library.

(int) `OaIsLibOpen(`*libname*`)`
Return 1 if the OpenAccess library named in the argument is open, 0 otherwise.

(int) `OaOpenLibrary(`*libname*`)`
Open the OpenAccess library of the given name, where the name should match a library defined in the `lib.defs` or `cds.lib` file. A library being open means that it is available for resolving undefined references when reading cell data in *Xic*. The return is 1 on success, 0 if error.

(int) `OaCloseLibrary(`*libname*`)`
Close the OpenAccess library of the given name, where the name should match a library defined in the `lib.defs` or `cds.lib` file. A library being open means that it is available for resolving undefined references when reading cell data in *Xic*. The return is 1 on success, 0 if error.

(int) `OaIsOaCell(`*libname*`,` *open_only*`)`
Return 1 if a cell with the given name can be resolved in an OpenAccess library, 0 otherwise. If the boolean value *open_only* is true, only open libraries are considered, otherwise all libraries are considered.

(int) `OaIsCellInLib(`*libname*`,` *cellname*`)`
Return 1 if the given cell can be found in the OpenAccess library given as the first argument, 0 otherwise.

(int) `OaIsCellView(`*cellname*`,` *viewname*`,` *open_only*`)`
Return 1 if the cellname and viewname resolve as a cellview in an OpenAccess library, 0 otherwise. If the boolean *open_only* is true, only open libraries are considered, otherwise all libraries are considered.

(int) `OaIsCellViewInLib(`*libname*`,` *cellname*`,` *viewname*`)`
Return 1 is the cellname and viewname resolve as a cellview in the given OpenAccess library, 0 otherwise.

(int) `OaCreateLibrary(`*libname*`,` *techlibname*`)`
This will create the library in the OpenAccess database if *libname* currently does not exist. This will also set up the technology for the new library if *techlibname* is given (not null or empty). The new library will attach to the same library as *techlibname*, or will attach to *techlibname* if it has a local tech database. If *techlibname* is given then it must exist.

(int) `OaBrandLibrary(`*libname*`)`
Set or remove the *Xic* "brand" of the given library. *Xic* can only write to a branded library. If the boolean *branded* is true, the library will have its flag set, otherwise the branded status is unset.

(int) `OaIsLibBranded(`*libname*`)`
Return 1 if the named library is "branded" (writable by *Xic*), 0 otherwise.

(int) `OaDestroy(`*libname*`,` *cellname*`,` *viewname*`)`
Destroy the named view from the given cell in the given OpenAccess library. If the *viewname* is null or empty, destroy all views from the named cell, i.e., the cell itself. If the *cellname* is null or empty, undefine the library in the library definition (`lib.defs` or `cds.lib`) file, and change the directory name to have a "`.defunct`" extension. We don't blow away the data, the user can revert by hand, or delete the directory.

(int) `OaLoad(`*libname*`,` *cellname*`)`
If *cellname* is null or empty, load all cells in the OpenAccess library named in *libname* into *Xic*. The current cell is not changed. Otherwise, load the cell and its hierarchy and make it the current cell. Whether the physical or electrical views are read, or both, is determined by the value of the OaUseOnly variable. If the value is "1" or starts with 'p' or 'P', only the physical (layout) views are read. If the value is "2" or starts with 'e' or 'E', only the electrical (schematic and symbol) views are read. If anything else or not set, both physical and electrical views are read.

(int) `OaReset()`
There is a table in *Xic* that records the cells that have been loaded from OpenAccess. This avoids the "merge control" pop-up which appears if a common subcell was previously read and is already in memory, the in-memory cell will not be overwritten. This function clears the table, and should be called if this protection should be ended, for example if the *Xic* database has been cleared.

(int) `OaSave(`*libname*`,` *allhier*`)`
Write the current cell to the OpenAccess library whose name is given in the first argument. This must exist, and be writable from *Xic*. Whether the physical or electrical views are written, or both, is determined by the value of the OaUseOnly variable. If the value is "1" or starts with 'p' or 'P', only the physical (layout) views are written. If the value is "2" or starts with 'e' or 'E', only the electrical (schematic and symbol) views are written. If anything else or not set, both physical and electrical views are written. The second argument is a boolean that if true (nonzero) indicates that the entire cell hierarchy under the current cell should be saved. Otherwise, only the current cell is saved.

The actual view names used are given in the OaDefLayoutView, OaDefSchematicView, and OaDefSymbolView variables, or default to "layout", "schematic", and "symbol".

(int) `OaAttachTech(`*libname*`,` *techlibname*`)`
If *techlibname* has an attached tech library, then that library will be attached to *libname*. If *techlibname* has a local tech database, then *techlibname* itself will be attached to *libname*. This will fail if *libname* has a local tech database. The local database should be destroyed first.

(string) `OaGetAttachedTech(`*libname*`)`
Return the name of the OpenAccess library providing the attached technology, or a null string if no attachment.

(int) `OaHasLocalTech(`*libname*`)`
Return 1 if the OpenAccess library has a local technology database, 0 if not.

(int) `OaHasLocalTech(`*libname*`)`
If the library does not have an attached or local technology database, create a new local database.

(int) `OaDestroyTech(`*libname*`,` *unattach_only*`)`
If *libname* has an attached technology library, unattach it. If the boolean second argument is false, and the library has a local database, destroy the database.

F.1.14 Mode

(int) `Mode(`*window*`,` *mode*`)`
This function switches *Xic* between physical and electrical modes, or switches sub-windows between the two viewing modes. The first argument is an integer 0–4, where 0 represents the main window, in which case the application mode is set, and 1–4 represent the sub-windows, in which case the viewing mode of that sub-window is set. The sub-window number is the same number as shown in the window title bar.

The second argument can be a number or a string. If a number and the nearest integer is not zero, the mode is electrical, otherwise physical. If a string that starts with '`e`' or '`E`', the mode is electrical, otherwise physical.

The return value is the new mode setting (0 or 1) or -1 if the indicated sub-window is not active.

(int) `CurMode(`*window*`)`
This function returns the current mode (physical or electrical) of the main window or sub-windows. The argument is an integer 0–4 where 0 represents the main window (and the application mode) and 1–4 represent sub-window viewing modes. The return value is 0 for physical mode, 1 for electrical mode, or -1 if the indicated sub-window does not exist. This function is identical to `GetWindowMode`.

F.1.15 Prompt Line

(int) `StuffText(`*string*`)`

The `StuffText` function stores the *string* in a buffer, which will be retrieved into the edit line on the next call to an editing function. The edit will terminate immediately, as if the user has typed *string*. Multiple lines can be stuffed, and will be retrieved in order. This function must be issued before the function which invokes the editor. Once a "stuffed" line is used, it is discarded.

(int) `TextCmd(`*string*`)`

This executes the command in string as if it were one of the keyboard "!" commands in *Xic*. The leading "!" is optional. Examples:

`TextCmd("!")`	brings up an xterm
`TextCmd("set ho deedo")`	sets variable 'ho'
`TextCmd("!select c")`	selects all subcells

(int) `GetLastPrompt()`

This function returns the most recent message that was shown on the prompt line, or would normally have been shown if *Xic* is not in graphics mode. Although the prompt line may have been erased, the last message is available until the next message is sent to the prompt line. The text on the prompt line while in edit mode is not saved and is not accessible with this function. An empty string is returned if there is no current message. This function never fails.

F.1.16 Scripts

(stringlist_handle) `ListFunctions()`

This function will re-read all of the `library` files in the script search path, and return a handle to a string list of the functions available from the libraries.

(untyped) `Exec(`*script*`)`

This function will execute a script. The argument is a string giving the script name or path. If the script is a file, it must have a "`.scr`" extension. The "`.scr`" extension is optional in the argument. If no path is given, the script will be opened from the search path or from the internal list of scripts read from the technology file or added with the **!script** command. If a path is given, that file will be executed, if found. It is also possible to reference a script which appears in a sub-menu of the **User Menu** by giving a modified path of the form "`@@/`*libname*`/.../`*scriptname*". The *libname* is the name of the script menu, the ... indicates more script menus if the menu is more than one deep, and the last component is the name of the script.

The return value is the result of the expression following "return" if a `return` statement caused termination of the script being executed. If the script did not terminate with a `return` statement with a following expression, the integer 1 is returned by `Exec`. If the script indicated by the argument to `Exec` could not be found, integer 0 is returned. If the `return` statement is used, the type of the return is determined by the type of object being returned.

Example: script1.scr

(*executable lines*)
```
return 3
```

in main script:

```
Print(Exec("script1")) # prints "3"
```

(int) `SetKey(`*password*`)`
This function sets the key used by *Xic* to decrypt encrypted scripts. The password must be the same as that used to encrypt the scripts. This function returns 1 on success, i.e., the key has been set, or 0 on failure, which shouldn't happen as even an empty string is a valid password.

(int) `HasPython()`
This function returns 1 if the Python language support plug-in has been successfully loaded, 0 otherwise.

(int) `RunPython(`*command* `[,` *arg*`, ...])`
Pass a command string to the Python interpreter for evaluation. The first argument is a path to a Python script file. Arguments that follow are concatenated and passed to the script. Presently, only string and scalar type arguments are accepted. The interpreter will have available the entire *Xic* scripting interface, though only the basic data types are useful. The Python interface description provides information about the header lines needed to instantiate the interface to *Xic* from Python (see 2.12).

This function exists only if the Python language support plug-in has been successfully loaded. The function returns 1 on success, 0 otherwise with an error message available from `GetError`.

(int) `RunPythonModuleFunc(`*module*`,` *function* `[,` *arg* `...])`
This function will call the Python interpreter, to execute the module function specified in the arguments. The first argument is the name of the module, which must be known to Python. The second argument is the name of the function within the module to evaluate. Following are zero or more function arguments, as required by the function.

This function exists only if the Python language support plug-in has been successfully loaded. The function returns 1 on success, 0 otherwise with an error message available from `GetError`.

(int) `ResetPython()`
Reset the Python interpreter. It is not clear that a user would ever need to call this.

This function exists only if the Python language support plug-in has been successfully loaded. The function always returns 1.

(int) `HasTcl()`
This function returns 1 if the Tcl language support plug-in was successfully loaded, 0 otherwise.

(int) `HasTk()`
This function returns 1 if the Tcl with Tk language support plug-in was successfully loaded, 0 otherwise.

(int) `RunTcl(`*command* `[,` *arg* `...])`
Pass a command string to the Tcl interpreter for evaluation. The first argument is a path to a Tck/Tk script. If both Tcl and Tk are available, the script file must have a `.tcl` or `.tk` extension. If only Tcl is available, there is no extension requirement, but the file should contain only Tcl commands. A Tcl script ie executed linearly and returns. A Tk script blocks, handling events until the last window is destroyed, at which time it returns.

Arguments that follow are concatenated and passed to the script. Presently, only string and scalar type arguments are accepted. The interpreter will have available the entire *Xic* scripting interface, though only the basic data types are useful. The Tcl/Tk interface description provides more information.

This function exists only if the Tcl language support plug-in has been successfully loaded. The function returns 1 on success, 0 otherwise with an error message available from `GetError`.

(int) `ResetTcl()`
Reset the Tcl/Tk interpreter. It is not clear that a user would ever need to call this.

This function exists only if the Tcl language support plug-in has been successfully loaded. The function always returns 1.

(int) `HasGlobalVariable(`*globvar*`)`
Return true if the passed string is the name of a global variable currently in scope. This is part of the exported global variable interface to Python and Tcl.

(int) `GetGlobalVariable(`*globvar*`)`
Return the value of the global variable whose name is passed. The function will generate a fatal error, halting the script, if the variable is not found, so one may need to check existence with `HasGlobalVariable`. The return type is the type of the variable, which can be any known type. This is for use in Python or Tcl scripts, providing access to the global variables maintained in the *Xic* script interpreter.

(int) `SetGlobalVariable(`*globvar*`,` *value*`)`
Set the value of the global variable named in the first argument. The function will generate a fatal error if the variable is not found, or the assignment fails due to type mismatch. This is for use in Python or Tcl scripts, providing access to the global variables maintained in the *Xic* script interpreter. Note that global variables can not be created from Python or Tcl, but values can be set with this function. Global variables can be used to return data to a top-level native script from a Tcl or Python sub-script.

F.1.17 Technology File

`GetTechName()`
This returns a string containing the current technology name, as set in the technology file with the `Technology` keyword.

(string) `GetTechExt()`
This returns a string containing the current technology file name extension.

(int) `SetTechExt(`*extension*`)`
This sets the current technology file extension to the string argument. It alters the name of new technology files created with the **Save Tech** button in the **Attributes Menu**.

(int) `TechParseLine(`*line*`)`
This function will parse and process a line of text is if read from a technology file. It can therefor modify parameters that are otherwise set in the technology file, after a technology file has been read, or if no technology file was read.

However, there are limitations.

1. There is no macro processing done on the line, it is parsed verbatim, and macro directives will not be understood.
2. There is no line continuation, all related text must appear in the given string.
3. The print driver block keywords are not recognized, nor are any other block forms, such as device blocks for extraction.
4. Layer block keywords are acceptable, however they must be given in a special format, which is

   [`elec`]`layer` *layername layer_block_line...*

i.e., the text must be prefaced by the `layer/eleclayer` keyword followed by an existing layer name. Note that new layers must be created first, before calling this function.

If the line is recognized and successfully processed, the function returns 1. Otherwise, 0 is returned, and a message is available from `GetError`.

(int) `TechGetFkeyString(`*fkeynum*`)`
This function returns the string which encodes the functional assignment of a function key. This is the same format as used in the technology file for the F1Key – F12Key keyword assignments. The argument is an integer with value 1–12 representing the function key number. The return value is a null string if the argument is out of range, or if no assignment has been made.

(int) `TechSetFkeyString(`*fkeynum*`,` *string*`)`
This function sets the string which encodes the functional assignment of a function key. This is the same format as used in the technology file for the F1Key – F12Key keyword assignments. The first argument is an integer with value 1–12 representing the function key number. The second argument is the string, or 0 to clear the assignment. The return value is 1 if an assignment was made, 0 if the first argument is out of range.

F.1.18 Variables

`Set(`*name*`,` *string*`)`
The `Set` function allows variable *name* to be set to *string* as with the **!set** keyboard operation in *Xic*. Some variables, such as the search paths, directly affect *Xic* operation. The `Set` function can also set arbitrary variables, which may be useful to the script programmer. To set a variable, both arguments should be strings. If the second argument is the constant zero (`0` or `NULL`, not `"0"`) or a null (not empty) string, the variable will be unset if set. As with **!set**, forms like $(*name*) are expanded. If *name* matches the name of a previously set variable, that variable's value string replaces the form. Otherwise, if *name* matches an environment variable, the environment variable text replaces the form.

The `Set` function will permanently change the variable value. See the `PushSet` function for an alternative.

`Unset(`*name*`)`
This function will unset the variable. No action is taken if the variable is not already set. This is equivalent to `Set`(*name*, 0).

`PushSet(`*name*`,` *string*`)`
This function is similar to `Set`, however the previous value is stored internally, and can be restored with `PopSet`. In addition, all variables set (or unset) with `PushSet` are reverted to original values when the script exits, thus avoiding permanent changes. There can be arbitrarily many `PushSet` and `PopSet` operations on a variable.

`PopSet(`*name*`)`
This reverts a variable set with `PushSet` to its previous state. If the variable has not been set (or unset) with `PushSet`, no action is taken.

(string) `SetExpand(`*string*`,` *use_env*`)`
This function returns a copy of *string* which expands variable references in the form $(*word*) in *string*. The *word* is expected to be a variable previously set with the `Set` function or **!set** command. The value of the variable replaces the reference in the returned string. If the integer *use_env* is nonzero, variables found in the environment will also be substituted. If *word* is not resolved, no change is made. Otherwise, in general, the token is replaced with the value of *word*.

There is an exception to the direct-substitution rule. If any substitution string is of the form "`(...)`", then the parentheses and leading/trailing white space are stripped before substitution, and the entire substituted string is enclosed in parentheses if it is not already. This is for convenience when adding a directory to a search path (see 2.6) variable, and the path is enclosed in parentheses. See the **!set** command description in 19.26 for more information.

(string) `Get(`*name*`)`
The `Get` function returns a string containing the value of *name*, which has been previously set with the `Set` function, or otherwise from within *Xic*. A null string is returned if the named variable has not been set.

`JoinLimits(`*flag*`)`
This is a convenience function to set/unset the variables which control the polygon joining process, i.e., JoinMaxPolyVerts, JoinMaxPolyQueue, and JoinMaxPolyGroup. If the argument is zero, each of these variables is set to zero, removing all limits. If the argument is nonzero, the variables are unset, meaning that the default limits will be applied. The default limits generally speed processing, but will often leave unjoined joinable pieces when complex polygons are constructed. The status of the variables will persist after the script terminates. This function has no return value.

F.1.19 *Xic* Version

(string) `VersionString()`
This function returns a string containing the current *Xic* version in a form like "`2.5.40`".

F.2 Main Functions 2

F.2.1 Arrays

(int) `ArrayDims(`*out_array*`,` *array*`)`
This function returns the size (number of storage locations) of an array, and possibly the size of each dimension. Arrays can have from one to three dimensions. If the first argument is an array with size three or larger, the size of each dimension of the array in the second argument is stored in the first three locations of the first argument array, with the 0'th index being the lowest order. Unused dimensions are saved as 0. If the first argument is an integer 0, no dimension size information is returned. The size of the array (number of storage locations, which should equal the product of the nonzero dimensions) is returned by the function.

(int) `ArrayDimension(`*out_array*`,` *array*`)`
This function is very similar to `ArrayDims`, and the arguments have the same types and purpose as for that function. The return value is the number of dimensions used (1–3) if the second argument is an array, 0 otherwise. Unlike `ArrayDims`, this function does not fail if the second argument is not an array.

(int) `GetDims(`*array*`,` *out_array*`)`
This is for backward compatibility. This function is equivalent to `ArrayDimension`, but the two arguments are in reverse order. This function may disappear – don't use.

(int) `DupArray(`*desc_array*`,` *src_array*`)`
This function duplicates the *src_array* into the *dest_array*. The *dest_array* argument must be an

unreferenced array. Upon successful return, the *dest_array* will be a copy of the *src_array*, and the return value is 1. If the *dest_array* can not be resized due to its being referenced by a pointer, 0 is returned. The function will fail if either argument is not an array.

(int) `SortArray(`*array*`, `*size*`, `*descend*`, `*indices*`)`
This function will sort the elements of the array passed as the first argument. The number of elements to sort is given in the second argument. The function will fail if *size* is negative, or will return without action if *size* is 0. The size is implicitly limited to the size of the array. The sorted values will be ascending if the third argument is 0, descending otherwise. The fourth argument, if nonzero, is an array which will be filled in with the index mapping applied to the array. For example, if array[5] is moved to array[0] during the sort, the value of indices[0] will be 5. This array will be resized if necessary, but the function will fail if resizing fails.

If the array being sorted is multi-dimensional, the sorting will use the internal linear order. The return value is the actual number of items sorted, which will be the value of size unless this was limited by the actual array size.

F.2.2 Bitwise Logic

All numerical data are stored internally in double-precision floating point representation. These functions convert the internal values to unsigned integer data, apply the operation, and return the floating-point representation of the result. This should be invisible to the user, but assumes well-behaved numerics in the host computer.

(unsigned int) `ShiftBits(`*bits*`, `*val*`)`
This function will shift the binary representation of the unsigned integer *bits* by the integer *val*. If *val* is positive, the bits are shifted to the right, or if negative the bits are shifted to the left. The function returns the shifted value.

(unsigned int) `AndBits(`*bits1*`, `*bits2*`)`
This function returns the bitwise AND of the two arguments, which are taken as unsigned integers.

(unsigned int) `OrBits(`*bits1*`, `*bits2*`)`
This function returns the bitwise OR of the two arguments, which are taken as unsigned integers.

(unsigned int) `XorBits(`*bits1*`, `*bits2*`)`
This function returns the bitwise exclusive-OR of the two arguments, which are taken as unsigned integers.

(unsigned int) `NotBits(`*bits*`)`
This function returns the bitwise NOT of the argument, which is taken as an unsigned integer.

F.2.3 Error Reporting

The following functions provide an interface to the *Xic* error reporting and logging system. The first two functions operate on the "message" which is a list of strings generated by errors encountered in function calls. Within *Xic*, the message may or may not be added to the error log, which is accessible via the functions below. Logged messages are included in the error log file, and will be displayed in a pop-up on-screen. If not added to the error log, the message may be displayed in another type of pop-up window, or on the prompt line, or may be placed in a conversion log file.

(string) `GetError()`
This returns the current error text. Error messages generated by an unsuccessful operation that opens, translates, or writes cells or manipulates the database, can be retrieved with this function for diagnostic purposes. This function should be called immediately after an error return is detected, since subsequent operations may clear or change the error text. If there are no recorded errors, a "no errors" string is returned. This function never fails and always returns a message string.

`AddError(`*string*`)`
This function will add a string to the current error message, which can be retrieved with `GetError`. This is useful for error reporting from user-defined functions. Any number of calls can be made, with the retrieved text consisting of a concatenation of the strings, with line termination added if necessary, in reverse order of the `AddError` calls. No other built-in function should be executed between calls to `AddError`, or between a call that generated an error and a call to `AddError`, as this will cause the second string to overwrite the first.

(int) `GetLogNumber()`
Return the integer index of the most recent error message dumped to the errors log file. The return value is 0 if there are no errors recorded in the file.

(string) `GetLogMessage(`*message_num*`)`
Return the error message string corresponding to the integer argument, as was appended to the errors log file. The 10 most recent error messages are available. If the argument is out of range, a null string is returned. The range is the current index to (not including) this index minus 10, or 0, whichever is larger.

(int) `AddLogMessage(`*string*`,` *error*`)`
Apply a new message to the error/warning log file. The second argument is a boolean which if nonzero will add the string as an error message, otherwise the message is added as a warning. The return value is the index assigned to the new message, or 0 if the string is empty or null.

F.2.4 Generic Handle Functions

The following functions take as an argument any type of handle, though some of these functions may do nothing if passed an inappropriate handle type. In particular, for functions that operate on lists, the following handle types are meaningful:

Object	**Handle Type**
string	stringlist_handle
object	object_handle
property	prpty_handle
device	device_handle
device contact	dev_contact_handle
subcircuit	subckt_handle
subcircuit contact	subc_contact_handle
terminal	terminal_handle

(int) `NumHandles()`
This returns the number of handles of all types currently in the hash table. It can be used as a check to make sure handles are being properly closed (and thus removed from the table) in the user's scripts.

(int) `HandleContent(`*handle*`)`

This function returns the number of objects currently referenced by the list-type handle passed as an argument. The return value is 1 for other types of handle. The return value is 0 for an empty or closed handle.

(int) `HandleTruncate(`*handle*`,` *count*`)`

This function truncates the list referenced by the handle, leaving the current item plus at most *count* additional items. If *count* is negative, it is taken as 0. The function returns 1 on success, or 0 if the handle does not reference a list or is not found.

(int) `HandleNext(`*handle*`)`

This function will advance the handle to reference the next element in its list, for handle types that reference a list. It has no effect on other handles. If there were no objects left in the list, or the handle was not found, 0 is returned, otherwise 1 is returned.

(handle) `HandleDup(`*handle*`)`

This function will duplicate a handle and its underlying reference or list of references. The new handle is not associated with the old, and should be iterated through or closed explicitly. For file descriptors, the return value is a duplicate descriptor to the underlying file, with the same read/write mode and file position as the original handle. If the function succeeds, a handle value is returned. If the function fails, 0 is returned.

(handle) `HandleDupNitems(`*handle*`,` *count*`)`

This function acts similarly to `HandleDup`, however for handles that are references to lists, the new handle will reference the current item plus at most *count* additional items. For handles that are not references to lists, the *count* argument is ignored. The new handle is returned on success, 0 is returned if there was an error.

(handle) `H(`*scalar*`)`

This function creates a handle from an integer variable. This is needed for using the handle values stored in the array created with the `HandleArray` function, or otherwise. Array elements are numeric variables, and can not be passed directly to functions expecting handles. This function performs the necessary data conversion.

Example:

```
SomeFunction(H(handle_array[3])).
```

Array elements are always numeric variables, though it is possible to assign a handle value to an array element. In order to use as a handle an array element so defined, the `H` function must be applied. Since scalar variables become handles when assigned from a handle, the `H` function should never be needed for scalar variables.

(int) `HandleArray(`*handle*`,` *array*`)`

This function will create a new handle for every object in the list referenced by the handle argument, and add that handle identifier to the array. Each new handle references a single object. The array argument is the name of a previously defined array variable. The array will be resized if necessary, if possible. It is not possible to resize an array referenced through a pointer, or an array with pointer references. The function returns 0 if the array cannot be resized and resizing is needed. The number of new handles is returned, which will be 0 if the handle argument is empty or does not reference a list. The handles in the array of handle identifiers can be closed conveniently with the `CloseArray` function. Since the array elements are numeric quantities and not handles, they can not be passed directly to functions expecting handles. The `H` function should be used to create a temporary handle variable from the array elements when a handle is needed: for example, `HandleNext(H(array[2]))`.

(int) `HandleCat(`*handle1*`,` *handle2*`)`
This function will add a copy of the list referenced by the second handle to the end of the list referenced by the first handle. Both arguments must be handles referencing lists of the same kind. The return value is nonzero for success, 0 otherwise.

(int) `HandleReverse(`*handle*`)`
This function will reverse the order of the list referenced by the handle. Calling this function on other types of handles does nothing. The function returns 1 if the action was successful, 0 otherwise.

(int) `HandlePurgeList(`*handle1*`,` *handle2*`)`
This function removes from the list referenced by the second handle any items that are also found in the list referenced by the first handle. Both handles must reference lists of the same type. The return value is 1 on success, 0 otherwise.

(int) `Close(`*handle*`)`
This function deletes and frees the handle. It can be used to free up resources when a handle is no longer in use. In particular, for file handles, the underlying file descriptor is closed by calling this function. The return value is 1 if the handle is closed successfully, 0 if the handle is not found in the internal hash table or some other error occurs.

(int) `CloseArray(`*array*`,` *size*`)`
This function will call `Close` on the first *size* elements of the array. The array is assumed to be an array of handles as returned from `HandleArray`. The function will fail if the *array* is not an array variable. The return value is always 1.

F.2.5 Memory Management

(int) `FreeArray(`*array*`)`
This function will delete the memory used in the *array*, and reallocate the size to 1. This function may be useful when memory is tight. It is not possible to free an array it there are variables that point to it. This function returns 1 on success, 0 otherwise.

(int) `CoreSize()`
This returns the total size of dynamically allocated memory used by *Xic*, in kilobytes.

F.2.6 Script Variables

(int) `Defined(`*variable*`)`
If a variable is referenced before it is assigned to, the variable has no type, but behaves in all ways as a string set to the variable's name. This function returns 1 if the argument has a type assigned, or 0 if it has no type.

(string) `TypeOf(`*variable*`)`
This function returns a string which indicates the type of variable passed as an argument. The possible returns are

“none”	variable has no type
“scalar”	variable is a scalar number
“complex”	variable is a complex number
“string”	variable is a string
“array”	variable is an array
“zoidlist”	variable is a zoidlist
“layer_expr”	variable is a layer_expr
“handle”	variable is a handle to something

F.2.7 Path Manipulation and Query

(int) `PathToEnd(`*path_name*`,` *dir*`)`
This function manipulates path strings. The string *path_name* can be anything, but it is usually one of “`Path`”, “`LibPath`”, “`HlpPath`”, or “`ScrPath`”, i.e., the name of a search path. The string *dir* will be appended to the path if it does not exist in the path, or is moved to the end if it does. If the *path_name* is not a recognized path keyword, a variable of that name will be created to hold the path. This can be used to store alternate paths.

(int) `PathToFront(`*path_name*`,` *dir*`)`
This is similar to the `PathToEnd` function, but the *dir* will be added or moved to the front of the path.

(int) `InPath(`*path_name*`,` *dir*`)`
This function returns 1 if *dir* is included in the path named in *path_name*, 0 otherwise.

(int) `RemovePath(`*path_name*`,` *dir*`)`
This function removes the directory *dir* from the search path, if it is present. The return value is 1 if the path was modified, 0 otherwise. The *path_name* argument has the same meaning as in `PathToEnd`.

F.2.8 Regular Expressions

(regex_handle) `RegCompile(`*regex*`,` *case_insens*`)`
This function returns a handle to a compiled regular expression, as given in the first (string) argument. The handle can be used for string comparison in `RegCompare`, and should be closed when no longer needed. The second argument is a flag; if nonzero the regular expression is compiled such that comparisons will be case-insensitive. If zero, the test will be case-sensitive. If the compilation fails, this function returns 0, and an error message can be obtained from `RegError`.

(int) `RegCompare(`*regex_handle*`,` *string*`,` *array*`)`
This function compares the regular expression represented by the handle to the string given in the second argument. If a match is found, the function returns 1, and the match location is set in the *array* argument, unless 0 is passed for this argument. If an array is passed, it must have size 2 or larger. The 0'th array element is set to the character index in the *string* where the match starts, and the next array location is set to the character index of the first character following the match. This function returns 0 if there is no match, and -1 if an error occurs. If -1 is returned, an error message can be obtained from `RegError`.

(string) `RegError(`*regex_handle*`)`
This function returns an error message string produced by the failure of `RegCompile` or `RegCompare`. It can be called after one of these functions returns an error value. The argument is the handle

value returned from `RegCompile`, which will be 0 if `RegCompile` fails. A null string is returned if the handle is bogus.

F.2.9 String List Handles

The following group of functions relate to lists of strings accessed by a handle. Such lists are returned by functions that find, for example, the list of layers in the current technology file, of a list of subcells in the current cell. Lists can also be defined by the user and are quite convenient for some purposes.

(stringlist_handle) `StringHandle(`*string*`,` *sepchars*`)`

This function returns a handle to a list of strings which are derived by splitting the *string* argument at characters found in the *sepchars* string. If *sepchars* is empty or null, the strings will be separated by white space, so each string in the handle list will be a word from the argument string.

(stringlist_handle) `ListHandle(`*arglist*`)`

This function creates a list of strings corresponding to the variable number of arguments, and returns a handle to the list. The arguments are converted to strings in the manner of the `Print` function, however each argument corresponds to a unique string in the list. The strings are accessed in (left to right) order of the arguments.

If no arguments are given, a handle to an empty list is returned. Calls to `ListAddFront` and/or `ListAddBack` can be used to add strings subsequently.

(string) `ListContent(`*stringlist_handle*`)`

This function returns the string currently referenced by the handle, and does *not* increment the handle to the next string in the list. If the handle is not found or contains no further list elements, a null string is returned. The function will fail if the handle is not a reference to a list of strings.

(int) `ListReverse(`*stringlist_handle*`)`

This function reverses the order of strings in the stringlist handle passed. If the operation succeeds the return value is 1, or if the list is empty or an error occurs the value is 0.

(string) `ListNext(`*stringlist_handle*`)`

This function will return the string at the front of the list referenced by the handle, and set the handle to reference the next string in the list. The function will fail if the handle is not a reference to a list of strings. A null string is returned if the handle is not found, or after all strings in the list have been returned.

(int) `ListAddFront(`*stringlist_handle*`,` *string*`)`

This function adds *string* to the front of the list of strings referenced by the handle, so that the handle immediately references the new string. The function will fail if the handle is not a reference to a string list, or the given string is null. The return value is 1 unless the handle is not found, in which case 0 is returned.

(int) `ListAddBack(`*stringlist_handle*`,` *string*`)`

This function adds *string* to the back of the list of strings referenced by the handle, so that the handle references the new string after all existing strings have been cycled. The function will fail if the handle is not a reference to a string list, or the given string is null. The return value is 1 unless the handle is not found, in which case 0 is returned.

(int) `ListAlphaSort(`*stringlist_handle*`)`

This function will alphabetically sort the list of strings referenced by the handle. The function will fail if the handle is not a reference to a list of strings. The return value is 1 unless the handle is not found, in which case 0 is returned.

(int) `ListUnique(`*stringlist_handle*`)`
This function deletes duplicate strings from the string list referenced by the handle, so that strings remaining in the list are unique. The function will fail if the handle is not a reference to a list of strings. The return value is 1 unless the handle is not found, in which case 0 is returned.

(string) `ListFormatCols(`*stringlist_handle*`,` *columns*`)`
This function returns a string which contains the column formatted list of strings referenced by the handle. The *columns* argument sets the page width in character columns. This function is useful for formatting lists of cell names, for example. The return is a null string if the handle is not found. The function fails if the handle does not reference a list of strings.

(string) `ListConcat(`*stringlist_handle*`,` *sepchars*`)`
This function returns a string consisting of each string in the list referenced by the handle separated by the *sepchars* string. If the *sepchars* string is empty or null, there is no separation between the strings. The function will fail if the handle does not reference a list of strings. A null string is returned if the handle is not found.

(int) `ListIncluded(`*stringlist_handle*`,` *string*`)`
This function compares *string* to each string in the list referenced by the handle and returns 1 if a match is found (case sensitive). If no match, or the handle is not found, 0 is returned. The function will fail if the handle is not a reference to a list of strings.

F.2.10 String Manipulation and Conversion

(string) `Strcat(`*string1*`,` *string2*`)`
This function appends *string2* to *string1* and returns the new string. The '+' operator is overloaded to also perform this function on string operands.

(int) `Strcmp(`*string1*`,` *string2*`)`
This function returns an integer representing the lexical difference between *string1* and *string2*. This is the same as the "`strcmp`" C library function, except that null strings are accepted and have the minimum lexical value. The comparison operators are overloaded to also perform this function on string operands.

(int) `Strncmp(`*string1*`,` *string2*`,` *n*`)`
This compares at most *n* characters in strings 1 and 2 and returns the lexical difference. This is equivalent to the C library "`strncmp`" function, except that null strings are accepted and have the minimum lexical value.

(int) `Strcasecmp(`*string1*`,` *string2*`)`
This internally converts strings 1 and 2 to lower case, and returns the lexical difference. This is equivalent to the C library "`strcasecmp`" function, except that null strings are accepted and have the minimum lexical value.

(int) `Strncasecmp(`*string1*`,` *string2*`,` *n*`)`
This internally converts strings 1 and 2 to lower case, and compares at most *n* characters, returning the lexical difference. This is equivalent to the C library "`strncasecmp`" function. except that null strings are accepted and have the minimum lexical value.

(string) `Strdup(`*string*`)`
This function returns a new string variable containing a copy of the argument's string. An error occurs if the argument is not string-type. Note that this differs from assignment, which propagates a pointer to the string data rather than copying.

(string) `Strtok(`*str*`,` *sep*`)`

The `Strtok` function is used to isolate sequential tokens in a string, *str*. These tokens are separated in the string by at least one of the characters in the string *sep*. The first time that `Strtok` is called, *str* should be specified; subsequent calls, wishing to obtain further tokens from the same string, should pass 0 instead. The separator string, *sep*, must be supplied each time, and may change between calls.

The `Strtok` function returns a reference to each subsequent token in the string, after replacing the separator character with a NULL character. When no more tokens remain, a null string is returned. Note that this is destructive to *str*.

This function is similar to the C library "`strtok`" function.

Example: print the space-separated words

```
teststr = "here are\tsome words"
word = Strtok(teststr, " \t")
Print("First word is", word);
while (word = Strtok(0, " \t"))
     Print("Next word:", word)
done
```

(string) `Strchr(`*string*`,` *char*`)`

The second argument is an integer representing a character. The return value is a pointer into *string* offset to point to the first instance of the character. If the character is not in the string, a null pointer is returned. This is basically the same as the C `strchr` function.

(string) `Strrchr(`*string*`,` *char*`)`

The second argument is an integer representing a character. The return value is a pointer into *string* offset to point to the last instance of the character. If the character is not in the string, a null pointer is returned. This is basically the same as the C `strrchr` function.

(string) `Strstr(`*string*`,` *char*`)`

The second argument is a string which is expected to be a substring of the string. The return value is a pointer into *string* to the start of the first occurrence of the substring. If there are no occurrences, a null pointer is returned. This is equivalent to the C `strstr` function.

(string) `Strpath(`*string*`)`

This returns a copy of the file name part of a full path given in the string.

(int) `Strlen(`*string*`)`

This function returns the number of characters in *string*.

(int) `Sizeof(`*arg*`)`

This function returns the allocated size of the argument, which is mostly useful for determining the size of an array. The return value is

string length	*arg* is a string
allocated array size	*arg* is an array
number of trapezoids	*arg* is a zoidlist
1	*arg* is none of above

(scalar) `ToReal(`*string*`)`

The returned value is a variable of type scalar containing the numeric value from the passed argument, which is a string. The text of the string should be interpretable as a numeric constant. If the argument is instead a scalar, the value is simply copied.

(string) `ToString(`*real*`)`
The returned value is a variable of type string containing a text representation of the passed variable, which is expected to be of type scalar. The format is the same as the C `printf` function with "`%g`" as a format specifier. If the argument is instead a string, the returned value points to that string.

(string) `ToStringA(`*real*`,` *digits*`)`
This will return a string containing the real number argument in SPICE format, which is a form consisting of a fixed point number followed by an alpha character or sequence which designates a scale factor. These are the same scale factors as used in the number parser. though "`mils`" is not used. The second argument is an integer giving the number of digits to print (in the range 2-15). If out of this range, a default of 6 is used.

If the first argument is a string, the string contents will be parsed as a number, and the result output as described above. If the parse fails, the number is silently taken as zero.

(string) `ToFormat(`*format*`,` *arg_list*`)`
This function returns a string, formatted in the manner of the C `printf` function. The first argument is a format string, as would be given to `printf`. Additional arguments (there can be zero or more) are the variables that correspond to the format specification. The type and position of the arguments must match the format specification, which means that the variables passed must resolve to strings or to numeric scalars. All of the formatting options described in the Unix manual page for `printf` are available, with the following exceptions:

1. No random argument access.
2. At most one '`*`' per substitution.
3. "`%p`" will always print zero.
4. "`%n`" is not supported.

The function fails if the first argument is not a string, is null, or there is a syntax error or unsupported construct, or there is a type or number mismatch between specification and arguments.

For example, the "id" returned from `GetObjectID` prints as a floating point value by default (since it is a large integer), which is usually not useful. One can print this as a hex value as follows:

```
id = GetObjectID(handle)
Print("Id =", ToFormat("0x%x", id))
```

(string) `ToChar(`*integer*`)`
This function takes as its input an integer value for a character, and returns a string containing a printable representation of the character. A null string is returned if the input is not a valid character index. This function can be used to preformat character data for printing with the various print functions.

F.2.11 Current Directory

`Cwd(`*path*`)`
This function changes the current working directory to that given by the argument. If *path* is null or empty, the change will be to the user's home directory. A tilde character ('~') appearing in *path* is expanded to the user's home directory as in a Unix shell. The return value is 1 if the change succeeds, 0 otherwise.

(string) `Pwd()`
This function returns a string containing the absolute path to the current directory.

F.2.12 Date and Time

(string) `DateString()`
This function returns a string containing the date and time in the format

```
Tue Jun 12 23:42:38 PDT 2001
```

(int) `Time()`
This returns a system time value, which can be converted to more useful output by `TimeToString` or `TimeToVals`. Actually, the returned value is the number of seconds since the start of the year 1970.

(int) `MakeTime(`*array*`,` *gmt*`)`
This function takes the time fields specified in the array and returns a time value is if returned from `Time`. If the boolean argument *gmt* is nonzero, the interpretation is GMT, otherwise local time. The array must be size 9 or larger, with the values set as when returned by the `TimeToVals` function (below).

Under Windows, the *gmt* argument is ignored and local time is used.

(string) `TimeToString(`*time*`,` *gmt*`)`
Given a time value as returned from `Time`, this returns a string in the form

```
Tue Jun 12 23:42:38 PDT 2001
```

If the boolean argument *gmt* is nonzero, GMT will be used, otherwise the local time is used.

(string) `TimeToVals(`*time*`,` *gmt*`,` *array*`)`
Given a time value as returned from `Time`, this breaks out the time/date into the array. The array must have size 9 or larger. If the boolean argument *gmt* is nonzero, GMT is used, otherwise local time is used.

The array values are set as follows.

array[0]	seconds (0 - 59).
array[1]	minutes (0 - 59).
array[2]	hours (0 - 23).
array[3]	day of month (1 - 31).
array[4]	month of year (0 - 11).
array[5]	year - 1900.
array[6]	day of week (Sunday = 0).
array[7]	day of year (0 - 365).
array[8]	1 if summer time is in effect, or 0.

The return value is a string containing an abbreviation of the local timezone name, except under Windows where the return is an empty string.

(int) `MilliSec()`
This returns the elapsed time in milliseconds since midnight January 1, 1970 GMT. This can be used to measure script execution time.

(int) `StartTiming(`*array*`)`
This will initialize the values in the array, which must have size 3 or larger, for later use by the `StopTiming` function. The return value is always 1.

(int) `StopTiming(`*array*`)`

This will place time differences (in seconds) into the array, since the last call to `StartTiming` (with the same argument). The array must have size 3 or larger. the components are:

0	Elapsed wall-clock time
1	Elapsed user time
2	Elapsed system time

The user time is the time the cpu spent executing in user mode. The system time is the time spent in the system executing on behalf of the process. This uses the UNIX `getrusage` or `times` system calls, which may not be available on all systems. If support is not available, e.g., in Windows, the user and system entries will be zero, but the wall-clock time is valid. This function always returns 1.

F.2.13 File System Interface

(string) `Glob(`*pattern*`)`

This function returns a string which is a filename expansion of the pattern string, in the manner of the C-shell. The pattern can contain the usual substitution characters `*`, `?`, `[ ]`, { }.

Example: Return a list of "`.gds`" files in the current directory.

```
list = Glob("*.gds")
```

(file_handle) `Open(`*file*`,` *mode*`)`

This function opens the file given as a string argument according to the string *mode*, and returns a file descriptor. The *mode* string should consist of a single character: '`r`' for reading, '`w`' to write, or '`a`' to append. If the returned value is negative, an error occurred.

(file_handle) `Popen(`*command*`,` *mode*`)`

This command opens a pipe to the shell command given as the first argument, and returns a file handle that can be used to read and/or write to the process. The handle should be closed with the `Close` function. This is a wrapper around the C library `popen` command so has the same limitations as the local version of that command. In particular, on some systems the mode may be reading or writing, but not both. The function will fail if either argument is null or if the `popen` call fails.

(file_handle) `Sopen(`*host*`,` *port*`)`

This function opens a "socket" which is a communications channel to the given *host* and *port*. If the *host* string is null or empty, the local host is assumed. The *port* number must be provided, there is no default. If the open is successful, the return value is an integer larger than zero and is a handle that can be used in any of the read/write functions that accept a file handle. The `Close` function should be called on the handle when the interaction is complete. If the connection fails, a negative number is returned. The function fails if there is a major error, such as no BSD sockets support.

(string) `ReadLine(`*maxlen*`,` *file_handle*`)`

The `ReadLine` function returns a string with length up to *maxlen* filled with characters read from *file_handle*. The *file_handle* must have been successfully opened for reading with a call to `Open`, `Popen`, or `Sopen`. The read is terminated by end of file, a return character, or a null byte. The terminating character is not included in the string. A null string is returned when the end of file is reached, or if the handle is not found. The function will fail if the handle is not a file handle, or *maxlen* is less than 1.

(int) `ReadChar(`*file_handle*`)`
The `ReadChar` function returns a single character read from *file_handle*, which must have been successfully opened for reading with an `Open`, `Popen`, or `Sopen` call. The function returns EOF (-1) when the end of file is reached, or if the handle is not found. The function will fail if the handle is not a file handle.

(int) `WriteLine(`*string*`, `*file_handle*`)`
The `WriteLine` function writes the content of *string* to *file_handle*, which must have been successfully opened for writing or appending with an `Open`, `Popen`, or `Sopen` call. The number of characters written is returned. The function will fail if the handle is not a file handle, or the *string* is null.

This function has the unusual property that it will accept the arguments in reverse order.

`WriteLine` does not append a carriage return character to the string. See the `PrintLog` function for a variable argument list alternative that does append a return character.

(int) `WriteChar(`*c, file_handle*`)`
This function writes a single character *c* to *file_handle*, which must have been successfully opened for writing or appending with a call to `Open`, `Popen`, or `Sopen`. The function returns 1 on success. The function will fail if the handle is not a file handle, or the integer value of *c* is not in the range 0–255.

This function has the unusual property that it will accept the arguments in reverse order.

(string) `TempFile(`*prefix*`)`
This function creates a unique temporary file name using the prefix string given, and arranges for the file of that name to be deleted when the program terminates. The file is not actually created. The return from this command is passed to the `Open` command to actually open the file for writing.

(stringlist_handle) `ListDirectory(`*path*`, `*filter*`)`
This function returns a handle to a list of names of files and/or directories in the given directory. If the *path* argument is null or empty, the current directory is understood. If the *filter* string is null or empty, all files and subdirectories will be listed. Otherwise the *filter* string can be "`f`" in which case only regular files will be listed, or "`d`" in which case only directories will be listed. If the directory does not exist or can't be read, 0 is returned, otherwise the return value is a handle to a list of strings.

(int) `MakeDir(`*path*`)`
This function will create a directory, if it doesn't already exist. If the *path* specifies a multi-component path, all parent directories needed will be created. The function will fail if a null or empty *path* is passed, otherwise the return value is 1 if no errors, 0 otherwise, with a message available from `GetError`. Passing the name of an existing directory is not an error.

(int) `FileStat(`*path*`, `*array*`)`
This function returns 1 if the file in *path* exists, and fills in some data about the file (or directory). If the file does not exist, 0 is returned, and the array is untouched.

The *array* must have size 7 or larger, or a value 0 can be passed for this argument. In this case, no statistics are returned, but the function return still indicates file existence.

If an array is passed and the path points to an existing file or directory, the array is filled in as follows:

array[0]
Set to 0 if *path* is a regular file. Set to 1 if *path* is a directory. Set to 2 if *path* is some other type of object.

array[1]
The size of the regular file in bytes, undefined if not a regular file.

array[2]
Set to 1 if the present process has read access to the file, 0 otherwise.

array[3]
Set to 1 if the present process has write access to the file, 0 otherwise.

array[4]
Set to 1 if the present process has execute permission to the file, 0 otherwise.

array[5]
Set to the user id of the file owner.

array[6]
Set to the last modification time. This is in a system-encoded form, use `TimeToString` or `TimeToVals` to convert.

(int) `DeleteFile(`*path*`)`
Delete the file or directory given in *path*. If a directory, it must be empty. If the file or directory does not exist or was successfully deleted, 1 is returned, otherwise 0 is returned with an error message available from `GetError`.

(int) `MoveFile(`*from_path*`, `*to_path*`)`
Move (rename) the file *from_path* to a new file *to_path*. On success, 1 is returned, otherwise 0 is returned with an error message available from `GetError`.

Except under Windows, directories can be moved as well, but only within the same file system.

(int) `CopyFile(`*from_path*`, `*to_path*`)`
Copy the file *from_path* to a new file *to_path*. On success, 1 is returned, otherwise 0 is returned with an error message available from `GetError`.

(int) `CreateBak(`*path*`)`
If the path file exists, rename it, suffixing the name with a "`.bak`" extension. If a file with this name already exists, it will be overwritten. The function returns 1 if the file was moved or doesn't exist, 0 otherwise, with an error message available from `GetError`.

(string) `Md5Digest(`*path*`)`
Return a string containing an MD5 digest for the file whose path is passed as the argument. This is the same digest as returned from the **!md5** command, and from the command

`openssl dgst -md5` *filepath*

available on many Linux-like systems.

If the file can not be opened, an empty string is returned, and an error message is available from `GetError`.

F.2.14 Socket and *Xic* Client/Server Interface

(string) `ReadData(`*size*`, `*skt_handle*`)`
This function will read exactly *size* bytes from a socket, and return string-type data containing the bytes read. The *skt_handle* must be a socket handle returned from `Sopen`. The function will fail (halt the script) only if the *size* argument is not an integer. On error, a null string is returned, and a message is available from `GetError`.

Note that the string can contain binary data, and if reading an ASCII string be sure to include the null termination byte. With binary data, the standard string manipulations may not work, and in fact can easily cause a program crash.

(string) `ReadReply(`*retcode*`,` *skt_handle*`)`

This function will read a response message from the *Xic* server. It expects the *Xic* server protocol and can not be used for other purposes.

The first argument is an array of size 3 or larger. Upon return, *retcode*[0] will contain the server return code, which is an integer 0–9, or possibly -1 on error. The value in *retcode*[1] will be the size of the message returned, which will be 0 or larger. The value in *retcode*[2] will be 0 on success, 1 on error. If an error occurred, an error message is available from `GetError`.

The return code in *retcode*[0] can have the following response types:

0	ok
1	in block, waiting for "end"
2	error
3	scalar data
4	string data
5	array data
6	zlist data
7	lexpr data
8	handle data
9	geometry data
-1	error reading data from server

The return value is of string-type, and may be null or binary. With binary data, the standard string manipulations may not work, and in fact can easily cause a program crash. It is not likely that the return will have any use other than as an argument to `ConvertReply`.

This function will fail (halt the script) only if the retcode argument is bad.

(variable) `ConvertReply(`*message*`,` *retcode*`)`

This function will parse and analyze a return message from the *Xic* server, which has been received with `ReadReply`. The first argument is the message returned from `ReadReply`. The second argument is an array of size 3 or larger, and can be the same array passed to `ReadReply`. The *retcode*[0] entry must be set to the message return code, and *retcode*[1] must be set to the size of the returned buffer. These are the same values as set in `ReadReply`.

Upon return, *retcode*[2] will contain a "data_ok" flag, which will be nonzero if the message contained data and the data were read properly. The function will fail (by halting the script) if the *retcode* argument is bad, i.e., not an array of size 3 or larger, or the *message* argument is not string-type.

The response codes 0–2 contain no data and are status responses from the server. The data responses will set the type and data of the function return, if successful. The *retcode*[2] value will be nonzero on success in these cases, and will always be false if "`longmode`" is not enabled.

Note that the type returned can be anything, and if assigned to a variable that already has a different type, an error will occur. The `delete` operator can be applied to the assigned-to variable to clear its state, before the function call.

The response type 9 is returned from the `geom` server function. This function will return a handle to a geometry stream, which can be passed to `GsReadObject`.

(int) `WriteMsg(`*string*`,` *skt_handle*`)`

This function will write a message to a socket, adding the proper network line termination. The first argument is a string containing the characters to write. The second argument is a socket handle obtained from `Sopen`. Any trailing line termination will be stripped from the string, and the network termination "`\r\n`" will be added.

This function never fails (halts the script). The return value is the number of bytes written, or 0 on error. On error, a message is available from `GetError`.

F.2.15 System Command Interface

(int) `Shell(`*command*`)`
The `Shell` function will execute *command* under an operating system shell. The *command* string consists of an executable name plus arguments, which should be meaningful to the operating system. The return value is the return code from the command, as obtained by the shell. The function will fail if the *command* string is null or empty.

(int) `System(`*command*`)`
This function sends the *command* string to the operating system for execution. This is an alias to the `Shell` function.

(int) `GetPID(`*parent*`)`
If the boolean argument is zero, this function returns the process ID of the currently running *Xic* process. If the argument is nonzero, the function returns the process ID of the parent process (typically a shell). The process ID is a unique integer assigned by the operating system.

F.2.16 Menu Buttons

(int) `SetButtonStatus(`*menu*`, `*button*`, `*set*`)`
This command sets the state of the specified button in the given menu or button array, which must be a toggle button. The button will be "pressed" if necessary to match the given state.

The first argument is a string giving the internal name of a menu. If the given name is null, empty, or "`main`", all of the menus in the main window will be searched. The internal menu names are as follows:

`main`	Main window menus
`side`	**Side Menu** buttons
`top`	**Top Menu** buttons
`sub1`	Wiewport 1 menus
`sub2`	Wiewport 2 menus
`sub3`	Wiewport 3 menus
`sub4`	Wiewport 4 menus
`file`	**File Menu**
`cell`	**Cell Menu**
`edit`	**Edit Menu**
`mod`	**Modify Menu**
`view`	**View Menu**
`attr`	**Attributes Menu**
`conv`	**Convert Menu**
`drc`	**DRC Menu**
`ext`	**Extract Menu**
`user`	**User Menu**
`help`	**Help Menu**

The second argument is the button name, which is the code name given in the tooltip window which pops up when the mouse pointer rests over the button. In the case of **User Menu** command buttons, the name is the text which appears on the button. Only buttons and menus visible in the current mode (electrical or physical) can be accessed.

It should be stressed that the string arguments refer to internal names, and *not* (in general) the label printed on the button. For a button, this is the five character or fewer name that is shown in the tooltip that pops up when the pointer is over the button. The same applies to the *menu* argument, however these names are not available from running *Xic*. The internal menu names are provided in the table above.

The identification of the menu is case insensitive. In the lower group of entries, only the first one or two characters have to match. Thus "Convert", "c", and "crazy" would all select the **Convert** menu, for example. One character is sufficient, except for 'e' (**Extract** and **Edit**). So, the menu argument can be the menu label, or the internal name, or some simplification at the user's discretion. For the upper group, the entire menu name must be given.

If the third argument is nonzero, the button will be pressed if it is not already engaged. If the third argument is zero, the button will be depressed if it is not already disengaged. The return value is 1 if the button state changed, 0 if the button state did not change, or -1 if the button was not found.

(int) `GetButtonStatus(`*menu*`,` *button*`)`
This command returns the status of the indicated menu button, which should be a toggle button. The two arguments are as described for `SetButtonStatus`. The return value is 1 if the button is engaged, 0 if the button is not engaged, or -1 if the button is not found.

(int) `PressButton(`*menu*`,` *button*`)`
This command "presses" the indicated button. This works with all buttons, toggle or otherwise, and is equivalent to clicking on the button with the mouse. The two arguments, which identify the menu and button, are described under `SetButtonStatus`. The return value is 1 if the button was pressed, 0 if the button was not found.

The following four functions send raw events to the window system. They are used primarily for the run time logging in the `xic_run.log` file. The run log consists entirely of executable statements, thus command scripts can be created by simply performing operations in *Xic*, and editing the `xic_run.log` file. Otherwise, these functions are not likely to be of much use to most *Xic* users.

`BtnDown(`*num*`,` *state*`,` *x*`,` *y*`,` *widget*`)`
This function generates a button press event dispatched to the widget specified by the last argument. The *num* is the button number: 1 for left, 2 for middle, 3 for right. The *state* is the "modifier" key state at the time of the event, and is the OR of 1 if **Shift** pressed, 4 if **Control** pressed, 8 if **Alt** pressed, as in X windows. Other flags may be given as per that spec, but are not used by *Xic*. The coordinates are relative to the window of the target, in pixels. The *widget* argument is a string containing a resource specifier for the widget relative to the application, the syntax of which is dependent upon the specific user interface. A call to `BtnDown` should be followed by a call to `BtnUp` on the same widget. There is no return value.

`BtnUp(`*num*`,` *state*`,` *x*`,` *y*`,` *widget*`)`
This function generates a button release event dispatched to the widget specified by the last argument. The *num* is the button number: 1 for left, 2 for middle, 3 for right. The *state* is the "modifier" key state at the time of the event, and is the OR of 1 if **Shift** pressed, 4 if **Control** pressed, 8 if **Alt** pressed, as in X windows. Other flags may be given as per that spec, but are not used by *Xic*. The coordinates are relative to the window of the target. The *widget* argument is a

string containing a resource path for the widget relative to the application, the syntax of which is dependent upon the specific user interface. A call to `BtnUp` should only follow a call to `BtnDown` on the same widget. There is no return value.

`KeyDown(`*keysym*`, `*state*`, `*widget*`)`
This function generates a key press event dispatched to the widget specified in the last argument. The *keysym* is a code representing the key te send. The *state* and *widget* arguments are as described for `BtnDown`. A call to `KeyDown` should followed by a call to `KeyUp`, on the same widget. There is no return value.

`KeyUp(`*keysym*`, `*state*`, `*widget*`)`
This function generates a key release event dispatched to the widget specified in the last argument. The *keysym* is a code representing the key te send. The *state* and *widget* arguments are as described for `BtnDown`. A call to `KeyUp` should only follow a call to `KeyDown`, on the same widget. There is no return value.

F.2.17 Mouse Input

(int) `Point(`*array*`)`
This function blocks until mouse button 1 (left button) is pressed, or the **Esc** key is pressed, while the pointer is in a drawing window. The coordinates of the pointer at the time of the press are returned in the array. The return value is 0 if **Esc** was pressed or 1 for a button 1 press. Buttons 2 and 3 have their normal effects while this function is active, i.e., they are not handled in this function.

Example:

```
a[2]
ShowPrompt("Click in a drawing window")
Point(a)
ShowPrompt("x=", a[0], "y=", a[1])
```

When a ghost image is displayed with the `ShowGhost` function, the coordinates returned are either snapped to the grid or not, depending on the mode number passed to `ShowGhost`. If no ghost image is displayed, the nearest grid point is returned.

If the `UseTransform` function has been called to enable use of the current transform, the current transform will be applied to the displayed objects when using mode 8. The translation supplied to `UseTransform` is ignored (the translation tracks the mouse pointer).

(int) `Selection()`
Block, but allow selections in drawing windows. Return on any keypress, or escape event. Return the number of selected objects in the selection list.

F.2.18 Graphical Input

(string) `PopUpInput(`*message*`, `*default*`, `*buttontext*`,, `*multiline*`)`
This function will pop up a text-input widget, into which the user can enter text. The function blocks until the user presses the affirmation button, at which time the text is returned, and the pop-up disappears. If the user instead presses the **Dismiss** button or otherwise destroys the pop-up, the script will halt.

The first argument is an explanatory string which is printed on the pop-up. If this argument is null or empty, a default message is used. Recall that passing 0 is equivalent to passing a null string.

The second argument is a string providing default text which appears in the entry area when the pop-up appears. If this argument is null or empty there will be no default text.

The third argument is a string giving text that will appear on the affirmation button. If null or empty, the button will show a default label.

The fourth argument is a boolean that when nonzero, a multi-line text input widget will be used. Otherwise, a single-line input widget will be used.

(int) `PopUpAffirm(`*message*`)`
This button pops up a small window which allows the user to answer yes or no to a question. Deleting the window is equivalent to answering no. The argument is a string which should contain the text to which the user responds. When the user responds, the pop-up disappears, and the return value is 1 if the user answered "yes", 0 otherwise.

(real) `PopUpNumeric(`*message*`,` *initval*`,` *minval*`,` *maxval*`,` *delta*`,` *numdgt*`)`
This function pops up a small window which contains a "spin button" for numerical entry. The user is able to enter a number directly, or by clicking on the increment/decrement buttons.

The first argument is a string providing explanatory text. The second argument provides the initial numeric value. The *minval* and *maxval* arguments are the minimum and maximum allowed values. The *delta* argument is the delta to increment or decrement when the user presses the up/down buttons. These parameters are all real values. The *numdgt* is an integer value which sets how many places to the right of a decimal point are shown.

If the user presses **Apply**, the pop-up disappears, and this function returns the current value. If the user presses the **Dismiss** button or otherwise destroys the widget, the script will halt.

F.2.19 Text Input

(scalar) `AskReal(`*prompt*`,` *default*`)`
The two arguments are both strings, or 0 (equivalent to the predefined constant `NULL`). The function will print the strings on the prompt line, and the user will type a response. The response is converted to a real number which is returned by the function. If either argument is null, that part of the message is not printed. The *prompt* is immutable, but the *default* can be edited by the user.

Example:

```
a = AskReal("enter a value for a ", "2.5")
```

(string) `AskString(`*prompt*`,` *default*`)`
The two arguments and the return value are strings. Similar to the `AskReal` function, however a string is returned.

Example:

```
title = AskString("Enter your title: ", "Senior Computer Geek")
```

(scalar) `AskConsoleReal(`*prompt*`,` *default*`)`
This function prompts the user for a number, in the console window. It is otherwise similar to the `AskReal` function.

(string) `AskConsoleString(`*prompt*`,` *default*`)`

This function prompts the user for a string, in the console window. It is otherwise similar to the `AskString` function.

(int) `GetKey()`

This function blocks until any key is pressed. The return value is a key code, which is system dependent, but is generally the "keysym" of the key pressed. If the value is less than 20, the value is an internal code.

F.2.20 Text Output

(string) `SepString(`*string*`,` *repeat*`)`

This function returns a string that is created by repeating the *string* argument *repeat* times. The *repeat* value is an integer in the range 1–132. The function will fail if *string* is null.

(int) `ShowPrompt(`*arg_list*`)`

Print the values of the arguments on the prompt line. The number of arguments is variable.

Example:

```
a = 2.5
b = "the value of a is "
ShowPrompt(b, a)
```

This code fragment will print "`the value of a is 2.5`" on the prompt line.

If given without arguments, the prompt line will be erased, but without disturbing the current message as returned with `GetLastPrompt`. The function returns 1 if something is printed (message updated), 0 otherwise.

(int) `SetIndent(`*level*`)`

This function sets the indentation level used for printing with the `Print` and `PrintLog` functions. The argument is an integer which specifies the column where printed output will start. The argument can also be a string in one of the following formats:

`"+`*N*`"`
: *N* is an optional integer (default 1), increases indentation by *N* columns.

`"-`*N*`"`
: *N* is an optional integer (default 1), decreases indentation by *N* columns.

`""`
: Empty string, does not change indentation.

The function returns the previous indentation level.

(int) `SetPrintLimits(`*num_array_elts*`,` *max_zoids*`)`

While printing with the `Print` family of functions, or when using `ListHandle`, the number of array points and trapezoids actually printed is limited. The default limits are 100 array points and 20 trapezoids. This function allows these limits to be changed. A value for either argument of -1 will remove any limit, 0 will keep the present limit, non-negative values will set the limit, and negative values of -2 or less will revert to the default values. This function always returns 1 and never fails.

(int) `Print(`*arg_list*`)`

This function will print the arguments on the console. This is the window from which *Xic* was launched. The number of arguments is variable. The printing is indented according to the level set with the `SetIndent` function.

Any type of variable can be printed. Handles will be printed as a string giving the handle type. For a zoidlist variable, the coordinates of the trapezoids are printed, one trapezoid per line, in order x-lower-left, x-lower-right, y-lower, x-upper-left, x-upper-right, y-upper. Arrays are printed as a sequence of numbers. The number of array elements and trapezoids printed is limited to 100 and 20, respectively, but these limits can be changed or removed with the `SetPrintLimits` function.

(int) `PrintLog(`*file_handle*`,` *arg_list*`)`
This works like the `Print` function, however output goes to a file previously opened for writing with the `Open` function. The first argument is the file handle returned from `Open`. Following arguments are printed to the file in order, using indentation set with the `SetIndent` function. The function returns the number of characters written. The function will fail if the handle is not a file handle.

(string) `PrintString(`*arg_list*`)`
This works like the `Print`, etc. functions, however it returns a string containing the text, and indentation as set with `SetIndent` is ignored.

(string) `PrintStringEsc(`*arg_list*`)`
This works exactly like `PrintString`, however, special characters in any string supplied as an argument are shown in their '\' escape form.

(int) `Message(`*arg_list*`)`
This function will print the arguments in a pop-up message window, indentation is ignored.

(int) `ErrorMsg(`*arg_list*`)`
This function will print the arguments in a pop-up error window, indentation is ignored.

(int) `TextWindow(`*fname*`,` *readonly*`)`
This function brings up a text editor window loaded with the file whose path is given in the *fname* string. If the integer *readonly* is 0, editing of the file is enabled, otherwise editing is prevented.

F.3 Main Functions 3

Many of the layer-related functions take a "standard layer argument". This can be an integer index number into the layer table, where the index is 1-based, and values less than 1 return the current layer. The argument can also be a string, giving a layer name in *layer*[`:`*purpose*] form, or an alias name. If the string is null or empty, the current layer is returned.

F.3.1 Grid and Edge Snapping

(int) `SetMfgGrid(`*mfg_grid*`)`
This will set the manufacturing grid to the value of the argument, provided that the value is in the range 0.0 – 100.0 microns. When the manufacturing grid is nonzero, the snap grid is constrained to integer multiples of the manufacturing grid. The function returns 1 if the argument is in range, in which case the value is accepted, 0 otherwise.

(real) `GetMfgGrid()`
This function returns the value of the manufacturing grid. When nonzero, the snap grid is constrained to integer multiples of the manufacturing grid.

(int) `SetGrid(`*interval*`, `*snap*`, `*win*`)`

This function sets the grid parameters for the window indicated by the third argument, which is 0 for the main window or 1–4 for the sub-windows. The interval argument sets snap grid spacing, in microns. This value can be zero, in which case the present value is retained.

The snap value is an integer in the range of -10 to 10. If positive, the number provides the number of snap grid intervals between fine grid lines. If negative, the absolute value is the number of fine grid lines displayed per snap grid interval. If zero, the present setting is retained.

For electrical mode windows, the snap points must be on multiples of one micron. If not, this function returns 0 and the grid is unchanged. The function also returns 0 if the window argument does not correspond to an existing window. The return is 1 if the operation succeeds.

The function does not redraw the window. The `Redraw()` function can be called to redraw the window if necessary.

(real) `GetGridInterval(`*win*`)`

This function returns the fine grid interval in microns for the grid in the window indicated by the argument, which is 0 for the main window or 1–4 for the sub-windows. The function returns 0 if the argument does not correspond to an existing window.

(real) `GetSnapInterval(`*win*`)`

This function returns the snap grid interval in microns for the grid in the window indicated by the argument, which is 0 for the main window or 1–4 for the sub-windows. The function returns 0 if the argument does not correspond to an existing window.

(int) `GetGridSnap(`*win*`)`

This function returns the snap number for the grid in the window specified by the argument, which is 0 for the main window or 1–4 for the sub-windows. The snap number determines the number of snap grid intervals between fine grid lines if positive, or fine grid lines per snap interval if negative. The function returns 0 if the argument does not correspond to an existing window.

(int) `ClipToGrid(`*coord*`, `*win*`)`

The first argument to this function is a coordinate in microns. The return value is the coordinate, in microns, snapped to the nearest snap point of the grid of the window given in the second argument. The second argument is 0 for the main window, or 1–4 for the sub-windows. The function fails if the window argument does not correspond to an existing window.

Note that this function must be called twice for an x,y coordinate pair. This function ignores the edge-snapping modes, only taking into account the grid resolution and snap values.

(int) `SetEdgeSnappingMode(`*win*`, `*mode*`)`

Change the edge snapping mode in a drawing window. The first argument is an integer representing the drawing window: 0 for the main window, and 1–4 for subwindows. The change will apply only to that window, though changes in the main window will apply to new sub-windows. The second argument is an integer in the range 0–2. The effects are

0	No edge snapping.
1	Edge snapping is enabled in some commands.
2	Edge snapping is always enabled.

The return value is 1 if the window edge snapping was updated, 0 otherwise.

(int) `SetEdgeOffGrid(`*win*`, `*off_grid*`)`

This will enable snapping to off-grid locations when edge snapping is enabled, in the given window. The first argument is an integer representing the drawing window: 0 for the main window, and

1–4 for subwindows. The second argument is a boolean which will allow off-grid snapping when true. The return value is 1 if the window parameter was updated, 0 otherwise.

(int) `SetEdgeNonManh(`*win*`,` *non_manh*`)`
This will enable snapping to non-Manhattan edges when edge snapping is enabled, in the given window. The first argument is an integer representing the drawing window: 0 for the main window, and 1–4 for subwindows. The second argument is a boolean which will allow snapping to non-Manhattan edges when true. The return value is 1 if the window parameter was updated, 0 otherwise.

(int) `SetEdgeWireEdge(`*win*`,` *wire_edge*`)`
This will enable snapping to wire edges when edge snapping is enabled, in the given window. The first argument is an integer representing the drawing window: 0 for the main window, and 1–4 for subwindows. The second argument is a boolean which will allow snapping to wire edges when true. The return value is 1 if the window parameter was updated, 0 otherwise.

(int) `SetEdgeWirePath(`*win*`,` *wire_path*`)`
This will enable snapping to the wire path when edge snapping is enabled, in the given window. The path is the set of line segments that invisibly run along the center of the displayed wire, which, along with the wire width and end style, actually defines the wire. The first argument is an integer representing the drawing window: 0 for the main window, and 1–4 for subwindows. The second argument is a boolean which will allow snapping to the wire path when true. The return value is 1 if the window parameter was updated, 0 otherwise.

(int) `GetEdgeSnappingMode(`*win*`)`
This function returns the edge snapping mode in effect for the given window. The argument is an integer representing the drawing window: 0 for the main window, and 1–4 for subwindows. The return value is -1 if the window is not found, 0-2 otherwise.

0	No edge snapping.
1	Edge snapping is enabled in some commands.
2	Edge snapping is always enabled.

(int) `GetEdgeOffGrid(`*win*`)`
This returns the setting of the allow off-grid edge snapping flag for the given window. The argument is an integer representing the drawing window: 0 for the main window, and 1-4 for subwindows. The return value is -1 if the window is not found, 0 or 1 otherwise tracking the state of the flag.

(int) `GetEdgeNonManh(`*win*`)`
This returns the setting of the allow non-Manhattan edge snapping flag for the given window. The argument is an integer representing the drawing window: 0 for the main window, and 1–4 for subwindows. The return value is -1 if the window is not found, 0 or 1 otherwise tracking the state of the flag.

(int) `GetEdgeWireEdge(`*win*`)`
This returns the setting of the allow wire-edge edge snapping flag for the given window. The argument is an integer representing the drawing window: 0 for the main window, and 1–4 for subwindows. The return value is -1 if the window is not found, 0 or 1 otherwise tracking the state of the flag.

(int) `GetEdgeWirePath(`*win*`)`
This returns the setting of the allow wire-path edge snapping flag for the given window. The argument is an integer representing the drawing window: 0 for the main window, and 1–4 for subwindows. The return value is -1 if the window is not found, 0 or 1 otherwise tracking the state of the flag.

(int) `SetRulerSnapToGrid(`*snap*`)`
This function sets the snap-to-grid behavior when creating rulers in the **Rulers** command. When set, the mouse cursor will snap to grid locations, otherwise not. In either case the cursor may snap to object edges if edge snapping is enabled. If the **Rulers** command is active the mode will change immediately, otherwise the new mode will apply when the command becomes active. The return value is 0 or 1 representing the previous flag value.

(int) `SetRulerEdgeSnappingMode(`*mode*`)`
This sets the edge snapping mode which is applied during the **Rulers** command. This command has its own default edge snapping state. This function changes only the initial state when the command starts, and will have no effect in a running command (use `SetEdgeSnappingMode` to alter the current setting). The argument is an integer 0–2.

0	No edge snapping.
1	Edge snapping is enabled in some commands.
2	Edge snapping is always enabled.

The function returns -1 if the argument is out of range, or 0–2 representing the previous state otherwise.

(int) `SetRulerEdgeOffGrid(`*off_grid*`)`
This sets the edge snapping allow off-grid flag which is applied during the **Rulers** command. This command has its own default edge snapping state. This function changes only the initial state when the command starts, and will have no effect in a running command (use `SetEdgeOffGrid` to alter the current setting). The argument is a boolean value which enables the flag when true.

The return value is 0 or 1 representing the previous flag state.

(int) `SetRulerEdgeNonManh(`*non_manh*`)`
This sets the edge snapping allow non-Manhattan flag which is applied during the **Rulers** command. This command has its own default edge snapping state. This function changes only the initial state when the command starts, and will have no effect in a running command (use `SetEdgeNonManh` to alter the current setting). The argument is a boolean value which enables the flag when true.

The return value is 0 or 1 representing the previous flag state.

(int) `SetRulerEdgeWireEdge(`*wire_edge*`)`
This sets the edge snapping allow wire-edge flag which is applied during the **Rulers** command. This command has its own default edge snapping state. This function changes only the initial state when the command starts, and will have no effect in a running command (use `SetEdgeWireEdge` to alter the current setting). The argument is a boolean value which enables the flag when true.

The return value is 0 or 1 representing the previous flag state.

(int) `SetRulerEdgeWirePath(`*wire_path*`)`
This sets the edge snapping allow wire-path flag which is applied during the **Rulers** command. This command has its own default edge snapping state. This function changes only the initial state when the command starts, and will have no effect in a running command (use `SetEdgeWirePath` to alter the current setting). The argument is a boolean value which enables the flag when true.

The return value is 0 or 1 representing the previous flag state.

(int) `GetRulerSnapToGrid()`
This returns the present default snap-to-grid state used during the **Rulers** command. The values are 0 or 1 depending on the state.

(int) `GetRulerEdgeSnappingMode()`
The return value is an integer 0-2 representing the default edge snapping mode to use during the **Rulers** command.

0	No edge snapping.
1	Edge snapping is enabled in some commands.
2	Edge snapping is always enabled.

(int) `GetRulerEdgeOffGrid()`
The return value is 0 or 1 depending on the setting of the edge snapping allow off-grid flag which is the default in the **Rulers** command.

(int) `GetRulerNonManh()`
The return value is 0 or 1 depending on the setting of the edge snapping allow non-Manhattan flag which is the default in the **Rulers** command.

(int) `GetRulerEdgeWireEdge()`
The return value is 0 or 1 depending on the setting of the edge snapping allow wire-edge flag which is the default in the **Rulers** command.

(int) `GetRulerEdgeWirePath()`
The return value is 0 or 1 depending on the setting of the edge snapping allow wire-path flag which is the default in the **Rulers** command.

F.3.2 Grid Style

(int) `ShowGrid(`*on*`,` *win*`)`
This function sets whether or not the grid is shown in a window. If the first argument is nonzero, the grid will be shown, otherwise the grid will not be shown. The second argument is an integer representing the drawing window: 0 for the main window, and 1–4 for sub-windows. The change will not be visible until the window is redrawn (one can call `Redraw`). If success, 1 is returned, or 0 is returned if the window does not exist.

(int) `ShowAxes(`*style*`,` *win*`)`
This function sets the axes presentation style in physical mode windows. The first argument is an integer 0–2, where 0 suppresses drawing of axes, 1 indicates plain axes, and 2 (or anything else) indicates axes with a box at the origin. The second argument is an integer representing the drawing window: 0 for the main window, 1–4 for sub-windows. Axes are never shown in electrical mode windows. On success, 1 is returned. If the window does not exist or is not showing a physical view, 0 is returned. The change will not be visible until the window is redrawn (one can call `Redraw`).

(int) `SetGridStyle(`*style*`,` *win*`)`
This function sets the line style used for grid rendering. The first argument is an integer mask that defines the on-off pattern. The pattern starts at the most significant '1' bit and continues through the least significant bit, and repeats. Set bits are rendered as the visible part of the pattern. If the style is 0, a dot is shown at each grid point. Passing -1 will give continuous lines. The second argument is an integer representing the drawing window: 0 for the main window, 1–4 for sub-windows. The function returns 1 on success, 0 if the window does not exist. The change will not be visible until the window is redrawn (one can call `Redraw`).

(int) `GetGridStyle(`*win*`)`
This function returns the line style mask used for rendering the grid in the given window. The mask has the interpretation described in the description of `SetGridStyle`. The argument is an

integer representing the window: 0 for the main window, and 1–4 for sub-windows. If the window does not exist, 0 is returned.

(int) `SetGridCrossSize(`*xsize*`, `*win*`)`
This applies only to grids with style 0 (dot grid). The *xsize* is an integer 0–6 which indicates the number of pixels to draw in the four compass directions around the central pixel. Thus, for nonzero values, the "dot" is rendered as a small cross. The second argument is an integer representing the drawing window: 0 for the main window, 1–4 for subwindows. The function returns 1 on success, 0 if the window does not exist or the style is nonzero. The change will not be visible until the window is redrawn (one can call `Redraw`).

(int) `GetGridCrossSize(`*win*`)`
This returns an integer 0–6, which will be nonzero only for grid style 0 (dot grid), and if the "dots" are being rendered as small crosses via a call to `SetGridCrossSize` or otherwise. The argument is an integer representing the window: 0 for the main window, and 1–4 for subwindows. If the window does not exist, 0 is returned.

(int) `SetGridOnTop(`*ontop*`, `*win*`)`
This function sets whether the grid is shown above or below rendered objects. If the first argument is nonzero, the grid will be shown above rendered objects. The second argument is an integer representing the drawing window: 0 for the main window and 1–4 for sub-windows. The function returns 1 on success, 0 if the window does not exist. The change will not be visible until the window is redrawn (one can call `Redraw`).

(int) `GetGridOnTop(`*win*`)`
This function returns 1 is the grid is shown on top of objects. The argument is an integer representing the drawing window: 0 for the main window and 1–4 for sub-windows. If the grid is shown below rendered objects, 0 is returned. If the window does not exist, -1 is returned.

(int) `SetGridCoarseMult(`*mult*`, `*win*`)`
This sets the number of fine grid lines per coarse grid line. The first argument is an integer 1–50 that provides this multiple (it is clipped to this range). If 1, the coarse grid color is used for all grid lines. The second argument represents the drawing window whose grid is being changed, 0 for the main drawing window, and 1–4 for sub-windows. The change will not be visible until the window is redrawn (one can call `Redraw()`).

The return value is 1 on success, 0 if the window does not exist.

(int) `GetGridCoarseMult(`*win*`)`
This returns the number of fine grid lines per coarse grid interval, as being used in the drawing window indicated by the argument. The argument is 0 for the main drawing window, 1–4 for sub-windows. If the window does not exist, zero is returned.

(int) `SaveGrid(`*regnum*`, `*win*`)`
This will save a grid parameter set to a register. The first argument is a register index value 0–7. Register 0 is used internally for the "last" value whenever grid parameters are changed, so is probably not a good choice unless this behavior is expected. These are the same registers as used with the **Grid Setup** panel, and are associated with the `PhysGridReg` and `ElecGridReg` keyword families in the technology file.

The second argument represents the drawing window whose grid parameters are to be saved. The value is 0 for the main drawing window, and 1–4 for sub-windows. Note that separate registers exist for electrical and physical mode, so register numbers can be reused in the two modes.

The return value is 1 on success, 0 if the indicated window does not exist, or the register value is out of range.

(int) `RecallGrid(`*regnum*`,` *win*`)`

This will recall a grid parameter set from a register, and update the grid of a drawing window. The first argument is a register index value 0–7. Register 0 is used internally for the "last" value whenever grid parameters are changed, so is probably not a good choice unless this behavior is expected. These are the same registers as used with the **Grid Setup** panel, and are associated with the `PhysGridReg` and `ElecGridReg` keyword families in the technology file.

The second argument represents the drawing window whose grid parameters are to be saved. The value is 0 for the main drawing window, and 1–4 for sub-windows. Note that separate registers exist for electrical and physical mode, so register numbers can be reused in the two modes.

The return value is 1 on success, 0 if the indicated window does not exist. The change will not be visible until the window is redrawn (one can call `Redraw()`).

F.3.3 Current Layer

(string) `GetCurLayer()`

This function returns a string containing the name of the current layer. If no current layer is defined, a null string is returned.

(int) `GetCurLayerIndex()`

This function returns the 1-based index of the current layer in the layer table. If no current layer is defined, 0 is returned.

(int) `SetCurLayer(`*stdlyr*`)`

This function sets the current layer as indicated by the standard layer argument. The return value is the 1-based index of the previous current layer in the layer table, or 0 if there was no current layer. This return can be passed as the argument to revert to the previous current layer.

(int) `SetCurLayerFast(`*stdlyr*`)`

This is like `GetCurLayer`, but there is no visible update, i.e., the layer table indication, and the current layer shown in various pop-ups, is unchanged. This is for speed when drawing. When drawing is finished, this should be called with the original current layer, or `SetCurLayer` should be called with some layer. The return value is the 1-based index of the previous current layer in the layer table, or 0 if there was no current layer. This return can be passed as the argument to revert to the previous current layer.

(int) `NewCurLayer(`*stdlyr*`)`

If the standard layer argument matches an existing layer, the current layer is set to that layer. Otherwise, a new layer is created, if possible, and the current layer is set to the new layer. The function will fail if it is not possible to create a new layer, for example if the name is not a valid layer name.

If the name is not in the *layer*`:`*purpose* form, any new layer created will use the default "`drawing`" purpose.

The return value is the 1-based index of the previous current layer in the layer table, or 0 if there was no current layer. This return can be passed as the argument to revert to the previous current layer.

(string) `GetCurLayerAlias()`

This function is deprecated, see `GetLayerAlias`. Return the alias name of the current layer, or a null string if there is no alias.

(int) `SetCurLayerAlias(`*alias*`)`
This function is deprecated, see `SetLayerAlias`. Set the alias name of the current layer. Returns 1 on success, 0 otherwise (possibly indicating a name clash).

(string) `GetCurLayerDescr()`
This function is deprecated, see `GetLayerDescr`. Return the description string of the current layer. This will be null if no description has been set.

(int) `SetCurLayerDescr(`*descr*`)`
This function is deprecated, see `SetLayerDescr`. Set the description string of the current layer. The return value is always 1.

F.3.4 Layer Table

(int) `LayersUsed()`
This returns a count of the layers in the layer table for the current display mode.

(int) `AddLayer(`*name*`, `*index*`)`
This adds the named layer to the layer table, in the position specified by the integer second argument. If the second argument is negative, the new layer will be added at the end, above all existing layers. If the index is 0, the new layer will be positioned at the index of the current layer, and the current layer and those above moved up. Otherwise, the index is a 1-based index into the layer table, where the new layer will be inserted. The layer at that index and those above will be moved up.

The name can match the name of an existing layer that has been removed from the layer table. It can also be a unique new name, and a new layer will be created. If the name matches an existing layer in the table, a new layer will also be created, but with an internally generated name.

The function will return 0 if it is not possible to create a new layer, for example if the name is not a valid layer name. On success 1 is returned.

If the name is not in the *layer*`:`*purpose* form, any new layer created will use the default "`drawing`" purpose.

(int) `RemoveLayer(`*stdlyr[]*`)`
This removes the layer indicated by the standard layer argument from the layer table if found. This returns 1 if the layer is found and removed, 0 otherwise.

(int) `RenameLayer(`*oldname*`, `*newname*`)`
The *oldname* is a standard layer argument. The *newname* is a string providing a new layer/purpose name in the *layer*[`:`*purpose*] form. If no purpose field is given, the default "`drawing`" purpose is assumed. This renames the layer specified in *oldname* to *newname*. The renamed layer will have any alias name removed.

This fails if *oldname* is unresolved or *newname* is null, and returns 0 on error, with an error message available from `GetError`.

(stringlist_handle) `LayerHandle(`*down*`)`
This function returns a handle to a list of the layer names from the layer table. If the argument is 0, the list is in ascending order. If the argument is nonzero, the list is in descending order. The layers used in the current display mode are listed.

(string) `GenLayers(`*stringlist_handle*`)`
This function returns a string containing a layer name from the layer table. The argument is the

handle returned by `LayerHandle`. A different layer is returned for each call. The null string is returned after all layers have been cycled through. This is equivalent to `ListNext`.

(stringlist_handle) `GetLayerPalette(`*regnum*`)`

The argument is an integer 0–7 corresponding to a layer palette register, as used with the **Layer Palette** panel, and associated with the `PhysLayerPalette` and `ElecLayerPalette` technology file keyword families. The return value is a stringlist handle, where the strings are the names of layers saved in the indexed palette register corresponding to the display mode of the main drawing window.

If the palette register is empty, or the argument is out of range, a scalar 0 is returned.

The register with index 0 is used internally to save the last **Layer Palette** user area before it pops down. Thus, this index should not be used unless this behavior is expected.

(int) `SetLayerPalette(`*list*`, `*regnum*`)`

The second argument is an integer 0–7 corresponding to a layer palette register, as used with the **Layer Palette** panel, and associated with the `PhysLayerPalette` and `ElecLayerPalette` technology file keyword families.

The first argument provides a list of layers, or null, to be saved in the indexed palette register corresponding to the display mode of the main drawing window. If the argument is a scalar 0, or a null string, the palette register will be cleared. Otherwise this argument can be a string consisting of space-separated layer names, or a stringlist handle, where the strings are layer names. The handle is unaffected by this function call.

The function returns 1 on success, 0 if the register index is out of range. The call will fail (halt the script) if a bad argument is passed.

There is no checking of the validity of the string saved as palette register data.

F.3.5 Layer Database

(int) `GetLayerNum(`*name*`)`

Return the component layer number given the component layer name. This is the *layer* part of the general *layer*[`:`*purpose*] layer name used in *Xic*. Each such name has a corresponding number in the database. If the name is not found, the return value is -1, which is reserved and is not a valid component layer number.

(string) `GetLayerName(`*num*`)`

Return the component layer name given the component layer number. If there is no name associated with the number, a null string is returned.

(int) `IsPurposeDefined(`*name*`)`

This returns 1 if the name matches a known purpose, 0 otherwise.

(int) `GetPurposeNum(name)`

This will return a purpose number associated with the name. If the name is not recognized, is null or empty, or matches "`drawing`" without case sensitivity, -1 is returned. This is the `drawing` purpose number.

(string) `GetPurposeName(`*num*`)`

Return a string giving the purpose name corresponding to the passed purpose number. If the purpose number is not recognized, or is the `drawing` purpose value of -1, a null string is returned.

F.3.6 Layers

(int) `GetLayerLayerNum(`*stdlyr*`)`
Return the component layer number associated with the layer indicated by the standard layer argument.

(int) `GetLayerPurposeNum(`*stdlyr*`)`
Return the purpose number associated with the layer indicated by the standard layer argument.

(string) `GetLayerAlias(`*stdlyr*`)`
This function returns a string containing the alias name of the layer indicated by the standard layer argument. The string will be null if no alias is set.

(int) `SetLayerAlias(`*stdlyr*`,` *alias*`)`
This function sets the alias name of the layer indicated by the standard layer first argument to the string given as the second argument, as for the `LppName` technology file keyword. The alias name is an optional secondary name for a layer/purpose pair. Most if not all functions that take a layer name argument will also accept an alias name.

The alias name will hide other layers if there is a name clash. This can be used for layer remapping, but the user must be careful with this. Layer name comparisons are case-insensitive.

Unlike the normal layer names, the alias name can have arbitrary punctuation, embedded white space, etc. However, leading and trailing white space is removed, and if the resulting string is empty or null, the existing alias name (if any) will be removed.

The function returns 1 if the alias name is applied to the layer, 0 if an error occurs. It is not possible to set the same name on more than one layer.

(string) `GetLayerDescr(`*stdlyr*`)`
This function returns a string containing the description of the layer indicated by the argument, which is a standard layer argument or derived layer name string. If no description has been set, a null string is returned.

(int) `SetLayerDescr(`*stdlyr*`,` *descr*`)`
This function sets the description of the layer indicated by the first argument, which is a standard layer argument or a derived layer name string, to the string given as the second argument. The description is an optional text string associated with the layer. The function always returns 1.

(int) `IsLayerDefined(`*name*`)`
The string argument contains a layer name. This can be the standard *layer*[`:`*purpose*] form, or can be an alias name. This function returns 1 if the argument can be resolved as the name of a layer in the layer table, in the current (electrical/physical) mode. If the layer can't be resolved, 0 is returned. The function will fail fatally if the argument is null or empty.

(int) `IsLayerVisible(`*stdlyr*`)`
The function returns 1 if the layer indicated by the argument, which is a standard layer argument or a derived layer name string, is currently visible (i.e., the visibility flag is set), 0 otherwise. If the layer is derived, the return is the flag status, derived layers are never actually visible.

(int) `SetLayerVisible(`*stdlyr*`,` *visible*`)`
This will set the visibility of the layer indicated in the first argument, which is a standard layer argument or a derived layer name string. The layer will be visible if the boolean second argument is nonzero, invisible otherwise. The previous visibility status is returned. If the layer is derived, the flag status is set, however derived layers are never visible.

(int) `IsLayerSelectable(`*stdlyr*`)`
The function returns 1 if the layer indicated by the argument, which is a standard layer argument or a derived layer name string. is currently selectable (i.e., the selectability flag is set), 0 otherwise.

(int) `SetLayerSelectable(`*stdlyr*`,` *selectable*`)`
This will set the selectability of the layer indicated in the first argument, which is a standard layer argument or a derived layer name string. The layer will be selectable if the boolean second argument is nonzero, not selectable otherwise. The previous selectability status is returned.

(int) `IsLayerSymbolic(`*stdlyr*`)`
The function returns 1 if the layer indicated by the argument, which is a standard layer argument or a derived layer name string, is currently symbolic (i.e., the `Symbolic` attribute is set), 0 otherwise.

(int) `SetLayerSymbolic(`*stdlyr*`,` *symbolic*`)`
This will set the `Symbolic` attribute of the layer indicated in the first argument, which is a standard layer argument or a derived layer name string. The layer will be symbolic if the boolean second argument is nonzero, not symbolic otherwise. The previous symbolic status is returned.

(int) `IsLayerNoMerge(`*stdlyr*`)`
The function returns 1 if the `NoMerge` attribute is set in the layer indicated by the argument, which is a standard layer argument or a derived layer name string, 0 otherwise.

(int) `SetLayerNoMerge(`*stdlyr*`,` *nomerge*`)`
This will set the `NoMerge` attribute of the layer indicated in the first argument, which is a standard layer argument or a derived layer name string. The layer will be given the `NoMerge` attribute if the boolean second argument is nonzero, or the attribute will be removed if present otherwise. The previous `NoMerge` status is returned.

(real) `GetLayerMinDimension(`*stdlyr*`)`
The return value is the `MinWidth` design rule value in microns for the layer indicated by the argument, which is a standard layer argument or a derived layer name string. If there is no `MinWidth` rule, or the DRC package is not available, 0 is returned.

(real) `GetLayerWireWidth(`*stdlyr*`)`
The function returns the default wire width for the layer indicated by the argument, which is a standard layer argument or a derived layer name string.

(int) `AddLayerGdsOutMap(`*stdlyr*`,` *layer_num*`,` *datatype*`)`
This function will add a mapping from the layer in the first argument (a standard layer argument or a derived layer name string) to the given GDSII layer number and data type. The layer number and data type are integers which define the layer in the GDSII world. When a GDSII file is written, the present layer will appear on the given layer number and data type in the GDSII file. It is possible to have multiple mappings of the layer, in which case the geometry from the named layer will appear on each layer number/data type given.

The function returns 1 on success, or 0 if the layer number or data type number is out of range. The acceptable range for the layer number and data type is [0 – 65535].

(int) `RemoveLayerGdsOutMap(`*stdlyr*`,` *layer_num*`,` *datatype*`)`
This function will remove a GDSII output layer mapping for the layer indicated in the first argument (a standard layer argument or a derived layer name string). The mapping may have been applied in the technology file, with the **Tech Parameter Editor** panel from the **Attributes Menu**, or by calling the `AddLayerGdsOutMap` function. The mappings removed match the given layer number and data type integers provided. These are in the range [-1 – 65535], where the value '-1' indicates a wild-card which will match all layer numbers or data types.

The return value is -1 if the layer number or data type is out of range. Otherwise, the return value is the number of mappings removed.

(int) `AddLayerGdsInMap(`*stdlyr*`,` *string*`)`
This function adds a GDSII input mapping record to the layer whose name is indicated in the first argument (a standard layer argument or a derived layer name string). The second argument is a string listing the layer numbers and data types which will map to the named layer, in the same syntax as used in the technology file. This is "*l1 l2-l3* ..., *d1 d2-d3* ...", where there are two comma separated fields. The left field consists of individual layer numbers and/or ranges of layer numbers, similarly the right field consists of individual data types and/or ranges of data types. Each field can have an arbitrary number of space-separated terms. For each layer listed or in a range, all of the data types listed or in a range will map to the named layer. There can be multiple input mappings applied to the named layer.

The function returns 0 if there was a syntax error. The function returns 1 if the mapping is successfully added.

(int) `ClearLayerGdsInMap(`*stdlyr*`)`
This function deletes all of the GDSII input mappings applied to the layer indicated in the argument, which is a standard layer argument or a derived layer name string. These mappings may have been applied through the technology file, added with the **Tech Parameter Editor** from the **Attributes Menu**, or added with the `AddLayerGdsInMap` function. This function returns 0 if the layer name does not exist in the symbol table for the current display mode (physical or electrical). Otherwise, the return value is the number of mapping records deleted.

(int) `SetLayerNoDRCdatatype(`*stdlyr*`,` *datatype*`)`
This function assigns a data type to be used for objects with the DRC skip flag set. The first argument is a standard layer argument indicating a physical layer or a derived layer name string. The second argument is the data type in the range [0 – 65535], or -1. If -1 is given, any previously defined data type is cleared. The function returns 0 if the layer name can't be resolved, or the data type is out of range. The value 1 is returned on success.

F.3.7 Layers – Extraction Support

These functions mainly support the extraction system, but are maintained in the main program and are therefor accepted in feature sets where the extraction system is disabled.

Many of the layer-related functions take a "standard layer argument". This can be an integer index number into the layer table, where the index is 1-based, and values less than 1 return the current layer. The argument can also be a string, giving a layer name in *layer*[`:`*purpose*] form, or an alias name. If the string is null or empty, the current layer is returned.

(string) `SetLayerExKeyword(`*stdlyr*`,` *string*`)`
The first argument is a standard layer argument indicating a physical layer, or a derived layer name string. The *string* argument is an extraction keyword and associated text, as would appear in a layer block in the technology file. The specification will be applied to the layer, overriding existing settings and possibly causing incompatible or redundant existing keywords to be deleted. This is similar to the editing functions of the **Tech Parameter Editor** from the **Attributes Menu**, when using the **Extract** or **Physical** pages.

The return is a status or error string, which may be null.

The following keywords can be specified:

```
Conductor
Routing
GroundPlane
GroundPlaneDark
GroundPlaneClear
TermDefault
Contact
Via
Dielectric
DarkField
Thickness
Rho
Sigma
Rsh
EpsRel
Capacitance
Lambda
Tline
Antenna
```

(string) `SetCurLayerExKeyword(`*string*`)`
This is similar to `SetLayerExKeyword`, but applies to the current layer. This function is deprecated and not recommended for use in new scripts.

(int) `RemoveLayerExKeyword(`*stdlyr*`,` *keyword*`)`
The first argument is a standard layer argument indicating a physical layer, or a derived layer name string. This will remove the specification for the extract keyword given in the argument from the layer. The argument must be one of the extraction keywords, i.e., those listed for `SetCurLayerExKeyword`. The return value is 1 if a specification was removed, 0 otherwise.

(int) `RemoveCurLayerExKeyword(`*keyword*`)`
This is similar to `RemoveLayerExKeyword` but applies to the current layer. This function is deprecated and not recommended for use in new scripts.

(int) `IsLayerConductor(`*stdlyr*`)`
The function returns 1 if the Conductor keyword is given or implied for the layer indicated by the argument, which is a standard layer argument or a derived layer name string, 0 otherwise.

(int) `IsLayerRouting(`*stdlyr*`)`
The function returns 1 if the Routing keyword is given for the layer indicated by the argument, which is astandard layer argument or a derived layer name string, 0 otherwise.

(int) `IsLayerGround(`*stdlyr*`)`
The function returns 1 if one of the GroundPlane keywords was given for the layer indicated by the argument, which is a standard layer argument or a derived layer name string, 0 otherwise.

(int) `IsLayerContact(`*stdlyr*`)`
The function returns 1 if the Contact keyword is given for the layer indicated by the argument, which is a standard layer argument or a derived layer name string, 0 otherwise.

(int) `IsLayerVia(`*stdlyr*`)`
The function returns 1 if the Via keyword is given for the layer indicated by the argument, which is a standard layer argument or a derived layer name string, 0 otherwise.

(int) `IsLayerViaCut(`*stdlyr*`)`
The function returns 1 if the ViaCut keyword is given for the layer indicated by the argument, which is a standard layer argument or a derived layer name string, 0 otherwise.

(int) `IsLayerDielectric(`*stdlyr*`)`
The function returns 1 if the Dielectric keyword is given for the layer indicated by the argument, which is a standard layer argument or a derived layer name string, 0 otherwise.

(int) `IsLayerDarkField(`*stdlyr*`)`
The function returns 1 if the DarkField keyword is given or implied for the layer indicated by the argument, which is a standard layer argument or a derived layer name string, 0 otherwise.

(real) `GetLayerThickness(`*stdlyr*`)`
The function returns the value of the Thickness parameter given for the layer indicated by the argument, which is a standard layer argument or a derived layer name string.

(real) `GetLayerRho(`*stdlyr*`)`
The function returns the resistivity in ohm-meters of the layer indicated by the argument, which is a standard layer argument or a derived layer name string, as given by the `Rho` or `Sigma` parameters, if given. If neither of these is given, and `Rsh` and `Thickness` are given, the return value will be `Rsh`*`Thickness`.

(real) `GetLayerResis(`*stdlyr*`)`
The function returns the sheet resistance for the layer indicated by the argument, which is a standard layer argument or a derived layer name string. This will be the value of the `Rsh` parameter, if given, or the values of `Rho/Thickness`, if `Rho` or `Sigma` and `Thickness` are given, or 0 if no value is available.

(real) `GetLayerTau(`*stdlyr*`)`
The function returns the Drude relaxation time for the layer indicated by the argument, which is a standard layer argument or a derived layer name string. This will be the value of the Tau parameter if given to the layer, 0 otherwise.

(real) `GetLayerEps(`*stdlyr*`)`
The function returns the relative dielectric constant for the layer indicated by the argument, which is a standard layer argument or derived layer name string, as given by the EpsRel parameter if applied.

(real) `GetLayerCap(`*stdlyr*`)`
The function returns the per-area capacitance for the layer indicated by the argument, which is a standard layer argument or a derived layer name string.

(real) `GetLayerCapPerim(`*stdlyr*`)`
The function returns the per-perimeter capacitance for the layer indicated by the argument, which is a standard layer argument or a derived layer name string.

(real) `GetLayerLambda(`*stdlyr*`)`
The function returns the value of the Lambda parameter for the layer indicated by the argument, which is a standard layer argument or a derived layer name string.

F.3.8 Selections

(int) `SetLayerSpecific(`*state*`)`
If the boolean state value is nonzero, all layers except for the current layer will become unselectable. Otherwise, all layers will be set to their default selectability state. The return value is always 1.

(int) `SetLayerSearchUp(`*state*`)`

This function will set layer-search-up selection mode (see 3.8.5) if the argument is nonzero, or normal mode otherwise. The return value is 1 or 0 representing the previous layer-search-up mode status.

(string) `SetSelectMode(`*ptr_mode*`, `*area_mode*`, `*sel_mode*`)`

This function allows the various selection modes to be set. These are the same modes that can be set with the **Selection Control Panel** provided by the **layer** button. If an input value is given as -1, that particular parameter will be unchanged. Otherwise, the possible values are

ptr_mode	*area_mode*	*sel_mode*
0 Normal	0 Normal	0 Normal
1 Select	1 Enclosed	1 Toggle
2 Modify	2 All	2 Add
		3 Remove

The return value is a string, where the first three characters are the previous values of *ptr_mode*, *area_mode*, and *sel_mode* as *integers*, not ASCII characters.

(int) `SetSelectTypes(`*string*`)`

This function allows setting of the object types that can be selected. This provides the default selection types, but does not apply to functions that provide an explicit argument for selection types.

The string argument consists of a sequence of characters whose presence indicates that the corresponding object type is selectable. These are:

`c`	cell instances
`b`	boxes
`p`	polygons
`w`	wires
`l`	labels

Other characters are ignored. If the string is null, empty, or contains none of the listed characters, all objects are enabled, as if the string "`cbpwl`" was entered.

This function always returns 1.

(int) `Select(`*left*`, `*bottom*`, `*right*`, `*top*`, `*types*`)`

This function performs a selection operation in the rectangle defined by the first four arguments (given in microns). The fifth argument is a string whose characters serve to enable selection of a given type of object: '`b`' for boxes, '`p`' for polygons, '`w`' for wires, '`l`' for labels, and '`c`' for instances. If this string is empty or null, then all objects will be selected. Any matching object that touches or overlaps the selection box will have its selection status toggled. For example,

```
Select(-INFINITY, -INFINITY, INFINITY, INFINITY, "c")
```

will select all subcells.

For more complex selections based on object types, etc., the `TextCmd` function can be used to call the **!select** command.

(int) `Deselect()`

This function deselects all selected objects.

F.3.9 Pseudo-Flat Generator

(object_handle) `FlatObjList(`*l*`, `*b*`, `*r*`, `*t*`, `*depth*`)`

This function provides access to the "pseudo-flat" object access functions that are part of internal DRC routines in *Xic*. This enables cycling through objects in the database without regard to the cell hierarchy. The first four arguments are the coordinates in microns of the bounding box to search in. The *depth* is the search depth, which can be an integer 0 or larger which sets the maximum depth to search (0 means search the current cell only, 1 means search the current cell plus the subcells, etc., and a negative integer sets the depth to search the entire hierarchy). This argument can also be a string starting with '`a`' such as "`a`" or "`all`" which indicates to search the entire hierarchy.

The return value is a list of box, polygon, and wire objects found in the given region on the current layer. Label and subcell objects are never returned. If *depth* is 0, the actual object pointers are returned in the list, and all of the object manipulation functions are available. Otherwise, the list references copies of the actual objects, transformed to the coordinate space of the current cell.

The copies of the objects can use substantial memory if the list is very long. The `FlatObjGen` function provides another access interface that can use less memory.

(handle) `FlatObjGen(`*l*`, `*b*`, `*r*`, `*t*`, `*depth*`)`

This function provides access to the "pseudo-flat" object access functions that are part of internal DRC routines in *Xic*. This enables cycling through objects in the database without regard to the cell hierarchy. The first four arguments are the coordinates in microns of the bounding box to search in. The *depth* is the search depth, which can be an integer 0 or larger which sets the maximum depth to search (0 means search the current cell only, 1 means search the current cell plus the subcells, etc., and a negative integer sets the depth to search the entire hierarchy). This argument can also be a string starting with '`a`' such as "`a`" or "`all`" which indicates to search the entire hierarchy.

Similar to `FlatObjList`, objects on the current layer are returned, but through an intermediate handle rather than through a list, which can require significant memory. This function returns a special handle which is passed to the `FlatGenNext` function to actually retrieve the objects. Although this handle can be passed to the generic handle functions, most of these functions will have no effect. `HandleContent` will return 1, or 0 if the handle is exhausted. `HandleNext` will advance to the next object without saving the object. The other functions will return 0 and do nothing. The `Close` function should be called to delete the handle unless the handle is iterated to completion with `FlatGenNext` or `HandleNext`.

If *depth* is 0, the object pointers returned from `FlatGenNext` represent the actual object, and all object manipulation functions are available. Otherwise, transformed copies of the actual objects are returned, and there are restrictions on the operations that can be performed (see F.5.4).

(handle) `FlatObjGenLayers(`*l*`, `*b*`, `*r*`, `*t*`, `*depth*`, `*layers*`)`

This function is very similar to `FlatObjGen`, however it returns objects from layers named in the *layers* string. If the string is null or empty, objects on all layers will be returned. Otherwise, the string is a space separated list of layer names. The names are expected to match layers in the current display mode. Names that do not match any layer are silently ignored, though the function fails if no layer can be recognized.

(object_handle) `FlatGenNext(`*handle*`)`

This takes as an argument the handle returned from `FlatObjGen` or `FlatObjGenLayers`, and returns an object handle which contains a single object returned from the generator. If the *depth* argument passed to these functions was nonzero, the objects are transformed copies. The returned handles should be closed after use by calling `Close`, or by calling an iterating function such as `HandleNext` or `ObjectNext`.

A new handle is returned for each call of this function, until no further objects are available in which case this function returns 0, and the handle passed as the argument will be closed.

(int) `FlatGenCount(`*handle*`)`
This function returns the number of objects that can be generated with the generator handle passed, which must be returned from `FlatObjGen` or `FlatObjGenLayers`. Generator handles do not cache an internal list of objects, so that the number of objects is unknown, which is why `HandleContent` returns 1 for generator handles. This function duplicates the generator context and iterates through the loop, counting returned objects. This can be an expensive operation.

(object_handle) `FlatOverlapList(`*object_handle*`,` *touch_ok*`,` *depth*`,` *layers*`)`
This function returns a handle to a list of objects that touch or overlap the object referenced by the *object_handle* argument. If *touch_ok* is nonzero, objects that touch but have zero overlap area will be included; if *touch_ok* is zero these objects will be skipped. The *depth* is the search depth, which can be an integer which sets the maximum depth to search (0 means search the current cell only, 1 means search the current cell plus the subcells, etc., and a negative integer sets the depth to search the entire hierarchy). This argument can also be a string starting with '`a`' such as "`a`" or "`all`" which indicates to search the entire hierarchy. If *depth* is not 0, the objects returned are transformed copies, otherwise the actual objects are returned. The *layer* argument is a string containing space-separated layer names of the layers to search for objects. If this is empty or null, all layers will be searched. The function fails if the handle argument is not a handle to an object list. The return value is a handle to a list of objects, or 0 if no overlapping or touching objects are found.

Only boxes, polygons, and wires are returned. The reference object can be any object. If the reference object is a subcell, objects from within the cell will be returned if *depth* is nonzero.

F.3.10 Geometry Measurement

(real) `Distance(`*x*`,` *y*`,` *x1*`,` *y1*`)`
This function computes the distance between two points, given in microns, returning the distance between the points in microns.

(real) `MinDistPointToSeg(`*x*`,` *y*`,` *x1*`,` *y1*`,` *x2*`,` *y2*`,` *aret*`)`
This function computes the shortest distance from *x,y* to the line segment defined by the next four arguments. The *aret* is an array of size at least 4, used for returned coordinates. If no return is needed, this argument can be set to 0. Upon return of a value greater than 0, the first two values in *aret* are *x* and *y*, the next two values are the point on the segment closest to *x,y*. All values are in microns.

(real) `MinDistPointToObj(`*x*`,` *y*`,` *object_handle*`,` *aret*`)`
This function computes the minimum distance from the point *x,y* to the boundary of the object given by the handle. The *aret* is an array of size at least 4 for return coordinates. If the return is not needed, this argument can be given as 0. Upon return of a value greater than 0, the first two values of aret will be *x* and *y*, the next two values will be the point on the boundary of the object closest to *x,y*. The function returns 0 if *x,y* touch or are enclosed in the object. The function will fail if the handle is not a reference to an object list. If there is an internal error, -1 is returned. All coordinates are in microns.

(real) `MinDistSegToObj(`*x1*`,` *y1*`,` *x2*`,` *y2*`,` *object_handle*`,` *aret*`)`
This function computes the minimum distance from the line segment defined by the first four

arguments to the boundary of the object given by the handle. The *aret* is an array of size at least 4 for return coordinates. If the return is not needed, this argument can be given as 0. Upon return of a value greater than 0, the first two values of *aret* will be the point on the line segment nearest the object, the next two values will be the point on the boundary of the object nearest to the line segment. The function returns 0 if the line segment touches or overlaps the object. The function will fail if the handle is not a reference to an object list. If there is an internal error, -1 is returned. All coordinates are in microns.

(real) `MinDistObjToObj(`*object_handle1*`,` *object_handle2*`,` *aret*`)`
This function computes the minimum distance between the two objects referenced by the handles. The *aret* is an array of size at least 4 for return coordinates. If the return is not needed, this argument can be given as 0. Upon return of a value greater than 0, the first two values of *aret* will be the point on the boundary of the first object nearest the second object, the next two values will be the point on the boundary of the second object nearest to the first object. The function returns 0 if the objects touch or overlap. The function will fail if either handle is not a reference to an object list. If there is an internal error, -1 is returned. All coordinates are in microns.

(real) `MaxDistPointToObj(`*x*`,` *y*`,` *object_handle*`,` *aret*`)`
This function finds the vertex of the object referenced by the handle farthest from the point *x,y* and returns this distance. The *aret* is an array of size at least 4 for return coordinates. If the return is not needed, this argument can be given as 0. Upon return of a value greater than 0, the first two values of aret will be *x* and *y*, the next two values will be the vertex of the object farthest from *x,y*. The function will fail if the handle is not a reference to an object list. If there is an internal error, -1 is returned. All coordinates are in microns.

(real) `MaxDistObjToObj(`*object_handle1*`,` *object_handle2*`,` *aret*`)`
This function finds the pair of vertices, one from each object, that are farthest apart. Both handles can be the same. The *aret* is an array of size at least 4 for return coordinates. If the return is not needed, this argument can be given as 0. Upon return of a value greater than 0, the first two values of *aret* will be the vertex from the first object, the next two values will be the vertex from the second object. The function will fail if either handle is not a reference to an object list. If there is an internal error, -1 is returned. All coordinates are in microns.

(int) `Intersect(`*object_handle1*`,` *object_handle2*`,` *touchok*`)`
This function determines whether the two objects referenced by the handles touch or overlap. The return value is 1 if the objects touch or overlap, 0 if the objects do not touch or overlap, or -1 if either handle points to an empty list or some other error occurred. The function fails if either handle is not a reference to an object list. If the *touchok* argument is nonzero, 1 will be returned if the objects touch but do not overlap. If touchok is 0, objects must overlap (have nonzero intersection area) for 1 to be returned.

F.4 Layout File Input/Output Functions

F.4.1 Layer Conversion Aliasing

There is provision for a layer aliasing mechanism which is applied when a data file is read. This capability is exported through an interface consisting of the UseLayerAlias and LayerAlias variables, and the script functions described below.

This is different from the `LppName` aliasing which applies to *Xic* layers, and is built into the layer database. The conversion aliases apply only while a layout file is being read.

(int) `ReadLayerCvAliases(`*handle_or_filename*`)`
The argument can be either a string giving a file name, or a file handle as returned from the `Open` function or equivalent (opened for reading). This function will read layer aliases, adding the definitions to the layer alias table. The format consists of lines of the form

name=*newname*

where both *name* and *newname* are four-character CIF-type layer names, and there is one definition per line. Lines with a syntax error or bad layer name are silently ignored. When the layer alias table is active, layers read from an input file will be substituted, i.e., if a layer named *name* is read, it will be replaced with *newname*. For data formats that use layer number and datatype numbers, such as GDSII, the layer names should be in the form of a four or eight-byte hex number, using upper case, where the left bytes represent the hex value of the layer number, zero padded, and the right bytes represent the zero padded datatype number. The eight-byte form should be used if the layer or datatype is larger than 255. Alternatively, the decimal form L,D is accepted for layer tokens, where the decimal layer and datatype numbers are separated by a comma with no space.

The function returns 1 on success, 0 otherwise.

(int) `DumpLayerCvAliases(`*handle_or_filename*`)`
The argument can be either a string giving a file name, or a file handle as returned from the `Open` function or equivalent (opened for writing). This function will dump the layer alias table. The format consists of lines of the form

name=*newname*

with one definition per line, where *name* and *newname* are CIF-type four character layer names, with *newname* being the replacement. The function returns 1 on success, 0 otherwise.

(int) `ClearLayerCvAliases()`
This function will remove all entries in the layer alias table. The function always returns 1.

(int) `AddLayerCvAlias(`*lname*`,` *new_lname*`)`
This function will add the layer name string *new_lname* as an alias for the layer name string *lname* to the layer alias table. If an error occurs, or an alias for *lname* already exists in the table (it will not be replaced) the function returns 0. The function otherwise returns 1.

(int) `RemoveLayerCvAlias(`*lname*`)`
This function removes any alias for `lname` from the layer alias table. The function always returns 1.

(string) `GetLayerCvAlias(`*lname*`)`
This function returns a string containing the alias for the passed layer name string, obtained from the layer alias table. If no alias exists for *lname*, a null string is returned.

F.4.2 Cell Name Mapping

(int) `SetMapToLower(`*state*`,` *rw*`)`
This function sets a flag which causes upper case cell names to be mapped to lower case when reading, writing, or format converting archive files. The first argument is a boolean value which if nonzero indicates case conversion will be applied, and if zero case conversion will be disabled.

The second argument is a boolean value that if zero indicates that case conversion will be applied when reading or format converting archive files, and nonzero will apply case conversion when writing an archive file from memory.

Within *Xic*, this flag can also be set from the panels available from the **Convert Menu**. The internal effect is to set or clear the InToLower or OutToLower variables. The return value is the previous setting of the variable.

(int) `SetMapToUpper(`*state*`,` *rw*`)`

This function sets a flag which causes lower case cell names to be mapped to upper case when reading, writing, or format converting archive files. The first argument is a boolean value which if nonzero indicates case conversion will be applied, and if zero case conversion will be disabled.

The second argument is a boolean value that if zero indicates that case conversion will be applied when reading or format converting archive files, and nonzero will apply case conversion when writing an archive file from memory.

Within *Xic*, this flag can also be set from the panels available from the **Convert Menu**. The internal effect is to set or clear the InToUpper or OutToUpper variables. The return value is the previous setting of the variable.

F.4.3 Cell Table

(int) `CellTabAdd(`*cellname*`,` *expand*`)`

This function is used to add cell names to the cell table for the current symbol table. The *cellname* must match a name in the global string table, which includes all cells read into memory or referenced by a CHD in memory.

If the boolean argument *expand* is nonzero, and the name matches a cell in the main database, the cell and all of the cells in its hierarchy will be added to the table, otherwise only the named cell will be added. It is not an error to add the same cell more than once, duplicates will be ignored.

If the *UseCellTab* variable is set, when a Cell Hierarchy Digest (CHD) is used to process a cell hierarchy for anything other than reading cells into the main database, cells listed in the cell table will override cells of the same name in the CHD. Thus, for example, one can substitute modified versions of cells as a layout file is being written.

The return value is 1 if all goes well, 0 if the table is not initialized or the cell is not found.

(int) `CellTabCheck(`*cellname*`)`

This function returns 1 if *cellname* is in the current cell table. If the *cellname* is valid but *cellname* is not in the table, 0 is returned. If the cellname is invalid (not a known cell name) or the cell table is uninitialized, the return value is -1.

(int) `CellTabRemove(`*cellname*`)`

If *cellname* is found in the current cell table, it will be removed. If the name was found in the table and removed, the return value is 1, otherwise the function returns 0.

(stringlist_handle) `CellTabList()`

This function returns a handle to a list of cell name strings obtained from the current cell table. If the table is empty, a scalar 0 is returned.

(int) `CellTabClear()`

This function will clear the current cell table. The function always returns 1.

F.4.4 Windowing and Flattening

(int) `SetConvertFlags(`*use_window*`,` *clip*`,` *flatten*`,` *ecf_level*`,` *rw*`)`

This function sets the status of flags used in format conversions and when writing output. The first three arguments correspond to the **Use Window**, **Clip to Window**, and **Flatten Hierarchy** buttons in the **Format Conversion** panel and similar. A nonzero integer value will set the flag, 0 will reset the flag.

The *ecf_level* is an integer 0–3 which sets the empty cell filtering level, as described for the **Format Conversion** panel in 14.10. The values are

0	No empty cell filtering.
1	Apply pre- and post-filtering.
2	Apply pre-filtering only.
3	Apply post-filtering only.

The *rw* argument is a boolean value that if zero indicates that the flags will be applied when converting archive files, as if set from the **Format Conversion** panel, and also apply to the `FromArchive` script function. With *rw* nonzero, the flags apply when writing output with the **Export Control** panel, or when using the `Export` and `To`*XXX* script functions. In this case, the *no_empties* flag is ignored, and the windowing is ignored except when flattening.

The data window can be set with the `SetConvertArea` script function. To apply clipping, both the *use_window* and *clip* flags must be set.

This function returns the previous value of the internal variable that contains the flags. The two ecf filter bits encode the filtering level as above. The bits are:

flatten	`0x1`
use_window	`0x2`
clip	`0x4`
ecf level0	`0x8`
ecf level1	`0x10`

(int) `SetConvertArea(`*l*`,` *b*`,` *r*`,` *t*`,` *rw*`)`

This function sets the rectangular area used to filter or clip objects during format conversion or file writing. The first four arguments are the window coordinates in microns, in the coordinate system of the top level cell, after scaling (if any).

The *rw* argument is a boolean value that if zero indicates that the values will be applied when converting archive files, as if set from the **Format Conversion** panel, and also apply when using the `FromArchive` script function. With *rw* nonzero, the values apply when writing output with the **Export Control** panel, or when using the `Export` and `To`*XXX* script functions. In this case, windowing is ignored except when flattening.

Use of the window can be enabled with the `SetConvertFlags` script function.

The function always returns 1.

F.4.5 Scale Factor

(real) `SetConvertScale(`*scale*`,` *which*`)`

This sets the scale used for conversions. There are three such scales, and the one to set is specified by the second argument, which is an integer 0–2.

which = 0

Set the scale used when converting an archive file directly to another format with the `FromArchive` script function or similar, or with the **Format Conversion** panel.

which = 1

Set the scale used when writing a file with the `Export` and `To`*XXX* script functions or similar, or the **Export Control** panel.

which = 2
Set the scale used when reading a file into *Xic* with the `Edit` or `OpenCell` functions or similar, or from the **Import Control** panel in *Xic*.

Script functions that read, write, or convert archive file data will in general make use of one of these scale factors, however if the function takes a scale value as an argument, that value will be used rather than the values set with this function.

The scale argument is a real value in the inclusive range 0.001 – 1000.0. The return value is the previous scale value.

F.4.6 Export Flags

(int) `SetStripForExport(`*state*`)`
This function sets the state of the **Strip For Export** flag. When set, output from the conversion functions will contain physical information only. This should be applied when generating output for mask fabrication. See the **Export Control** panel description for more information. If the integer argument is nonzero, the state will be set active. The return value is the previous state of the flag.

(int) `SetSkipInvisLayers(`*code*`)`
This function sets the variable which controls how invisible layers are treated by the output conversion functions. Layer visibility is set by clicking in the layer table with mouse button 2, or through the `SetLayerVisible` script function. If *code* is 0 or negative, invisible layers will be converted. If *code* is 1, invisible physical layers will not be converted. If *code* is 2, invisible electrical layers will not be converted. if *code* is 3 or larger, both electrical and physical invisible layers will not be converted. The return value is the previous code, which represents the state of the SkipInvisible variable, and the check boxes in the **Export Control** panel.

F.4.7 Import Flags

(int) `SetMergeInRead(`*state*`)`
This function controls the setting of an internal flag which enables merging of boxes and coincident objects while a file is being read. This flag is set from within *Xic* in the **Import Control** panel. If the integer argument is nonzero, the flag will be set. The return value is the previous state of the flag.

F.4.8 layout File Format Conversion

(int) `FromArchive(`*file_or_chd*`,` *destination*`)`
This function will read an archive (GDSII, CIF, CGX, or OASIS) file and translate the contents to another format. The *file_or_chd* argument is a string giving a path to the source archive file, or the name of a Cell Hierarchy Digest (CHD) in memory.

The type of file written is implied by the *destination*. If the *destination* is null or empty, native cell files will be created in the current directory. If the *destination* is the name of an existing directory, native cell files will be created in that directory. Otherwise, the extension of the *destination* determines the file type:

CGX	`.cgx`
CIF	`.cif`
GDSII	`.gds, .str, .strm, .stream`
OASIS	`.oas`

Only these extensions are recognized, however CGX and GDSII allow an additional `.gz` which will imply compression.

See the table in 18.10 for the features that apply during a call to this function.

The value 1 is returned on success, 0 otherwise, with possibly an error message available from `GetError`.

(int) `FromTxt(`*text_file*`,` *gds_file*`)`
This function will translate a text file in the format produced by the `ToTxt` function into a GDSII format file. This is useful after text mode editing has been performed on the file, to repair corruption or incompatibilities. If *gds_file* is null or empty, the name is generated from the *text_file* and given a "`.gds`" suffix.

(int) `FromNative(`*dir_path*`,` *archive_file*`)`
This function will translate native cell files found in the directory given in *dir_path* into an archive file given in the second argument. The format of the archive file produced is determined by the file extension provided, as for the `FromArchive` function. All native cell files found in the directory, except those with a "`.bak`" extension or whose name is the same as a device library symbol, are translated and concatenated, independently of any hierarchical relationship between the cells.

See the table in 18.10 for the features that apply during a call to this function. The supported manipulations are cell name aliasing, layer filtering, and scaling. Windowing manipulations and flattening are not supported. If a file named "`aliases.alias`" exists in the *dir_path*, it will be used as an input alias list for conversion. Each line consists of a native cell name followed by an alias to be used in the archive file, separated by white space.

The value 1 is returned on success, 0 otherwise, with possibly an error message available from `GetError`.

F.4.9 Export Layout File

(int) `SaveCellAsNative(`*cellname*`,` *directory*`)`
Save the cell named in the first (string) argument, which must exist in the current symbol table, to a native format file in the *directory*. If the directory string is null or empty (or 0 is passed for this argument), the cell is saved in the current directory.

See the table in 14.1 for the features that apply during a call to this function.

This functions returns 1 on success, 0 otherwise, with an error message likely available from `GetError`.

(int) `Export(`*filepath*`,` *allcells*`)`
This function exports design data to a disk file (or files). It can perform the same operations as the `To`*XXX* functions also described in this section. The type of file produced is set by the extension found on the *filepath* string. Recognized extensions are

native	`.xic`
CGX	`.cgx`
CIF	`.cif`
GDSII	`.gds, .str, .strm, .stream`
OASIS	`.oas`

Only these extensions are recognized, however CGX and GDSII allow an additional "`.gz`" which will imply compression. For native cell file output, the *filepath* must provide a path to an existing directory. If none of the other formats is matched, and the *filepath* exists as a directory, then native cell files will be written to that directory. Alternatively, if the *filepath* has a "`.xic`" extension, and the *filepath* with the `.xic` stripped is an existing directory, or the *filepath* including the `.xic` is an existing directory (checked in this order), again native cell files will be written to that directory.

The second argument is a boolean. If false, then the current cell hierarchy is written to output. If true, all cells found in the current symbol table will be written to output. In either case, by default cells that are sub-masters or library cells are not written unless the controlling variables are set, as from the **Export Control** panel. The other controls for windowing, flattening, scaling, and cell name mapping found in this panel apply as well, as do their underlying variables. These flags and values can also be set with the `SetConvertFlags`, `SetConvertArea`, and `SetConvertScale` functions, and others that apply to output generation. When writing all files, any windowing or flattening in force is ignored.

See the table in 14.1 for the features that apply during a call to this function.

The function return 1 on success, 0 otherwise with an error message available from `GetError`.

(int) `ToXIC(`*destination_dir*`)`
The `ToXIC` function will write the current cell hierarchy to disk files in native format, no questions asked. The argument is the directory where the *Xic* files will be created. If this argument is a null or empty string or zero, the *Xic* files will be created in the current directory.

See the table in 14.1 for the features that apply during a call to this function.

This functions returns 1 on success, 0 otherwise, with an error message likely available from `GetError`.

(int) `ToCGX(`*cgx_name*`)`
This function will write the current cell hierarchy to a CGX format file on disk. The argument is the name of the CGX file to create. If the *cgx_name* is null or an empty string, the name used will be the top level cell name suffixed with "`.cgx`".

See the table in 14.1 for the features that apply during a call to this function.

This functions returns 1 on success, 0 otherwise, with an error message likely available from `GetError`.

(int) `ToCIF(`*cif_name*`)`
This function will write the current cell hierarchy to a CIF format file on disk. The argument is the name of the CIF file to create. If the *cif_name* is null or an empty string, the name used will be the top level cell name suffixed with "`.cif`".

See the table in 14.1 for the features that apply during a call to this function.

This functions returns 1 on success, 0 otherwise, with an error message likely available from `GetError`.

(int) `ToGDS(`*gds_name*`)`
This function will write the current cell hierarchy to a GDSII format file on disk. The argument is the name of the GDSII file to create. If the *gds_name* is null or an empty string, the name used will be the top level cell name suffixed with "`.gds`".

See the table in 14.1 for the features that apply during a call to this function.

This functions returns 1 on success, 0 otherwise, with an error message likely available from `GetError`.

(int) `ToGdsLibrary(`*gds_name*`,` *cellname_list*`)`

This function will create a GDSII file from a list of cells in memory. The first argument is the name of the GDSII file to create. The second argument is a string consisting of space-separated cell names. The cells must be in memory, in the current symbol table. Both arguments must provide values as there are no defaults. The GDSII file will contain the hierarchy under each cell given, but any cell is added once only. The resulting file will in general contain multiple top-level cells.

See the table in 14.1 for the features that apply during a call to this function.

This functions returns 1 on success, 0 otherwise, with an error message likely available from `GetError`.

(int) `ToOASIS(`*oas_name*`)`

This function will write the current cell hierarchy to an OASIS format file on disk. The argument is the name of the OASIS file to create. If the *oas_name* is null or an empty string, the name used will be the top level cell name suffixed with "`.oas`".

See the table in 14.1 for the features that apply during a call to this function.

This functions returns 1 on success, 0 otherwise, with an error message likely available from `GetError`.

(int) `ToTxt(`*archive_file*`,` *text_file*`)`

This function will create an ASCII text file *text_file* from the contents of the archive file. The human-readable text file is useful for diagnostics. If *text_file* is null or empty, the name is derived from the *archive_file* and given a "`.txt`" extension. No output is produced for CIF, since these are already in readable format.

The third argument is a string, which can be passed to specify the range of the conversion. If this argument is passed 0, or the string is null or empty, the entire archive file will be converted. The string is in the form

[*start_offs*[`-`*end_offs*]] [`-r` *rec_count*] [`-c` *cell_count*]

The square brackets indicate optional terms. The meanings are

start_offs
An integer, in decimal or "`0x`" hex format (a hex integer preceded by "`0x`"). The printing will begin at the first record with offset greater than or equal to this value.

end_offs
An integer in decimal or "`0x`" hex format. If this value is greater than *start_offs*, the last record printed is at most the one containing this offset. If given, this should appear after a '`-`' character following the *start_offs*, with no space.

rec_count
A positive integer, at most this many records will be printed.

cell_count
A non-negative integer, at most the records for this many cell definitions will be printed. If given as 0, the records from the *start_offs* to the next cell definition will be printed.

See the table in 14.1 for the features that apply during a call to this function.

The function returns 1 on success, 0 otherwise with an error message possibly available from `GetError`.

F.4.10 Cell Hierarchy Digest

The Cell Hierarchy Digest (CHD) is a data structure for saving a description of a cell hierarchy in compact form. The CHD can be used to access data in the original file, without having to load the file, in an efficient manner. This capability is accessible from a set of script functions described below. This capability applies to physical data only.

(string) `FileInfo(`*filename*`,` *handle_or_filename*`,` *flags*`)`

This function provides information about the archive file given by the first argument. If the second argument is a string giving the name of a file, output will go to that file. If the second argument is a handle returned from the `Open` function or similar (opened for writing), output goes to the handle stream. In either case, the return value is a null string. If the second argument is a scalar 0, the output will be in the form of a string which is returned.

The third argument is an integer or string which determines the type of information to return. If an integer, the bits are flags that control the possible data fields and printing modes. The string form is a space or comma-separated list of text tokens or hex integers. The hex numbers or equivalent values for the text tokens are or'ed together to form the flags integer.

This is really just a convenience wrapper around the `ChdInfo` function. See the description of that function for a description of the flags. In this function, the following keyword flags will show as follows:

`alias`
No aliasing is applied.

`flags`
The flags will always be 0.

On error, a null string is returned, with an error message likely available from `GetError`.

(chd_name) `OpenCellHierDigest(`*filename*`,` *info_saved*`)`

This function returns an access name to a new Cell Hierarchy Digest (CHD), obtained from the archive file given as the argument. The new CHD will be listed in the **Cell Hierarchy Digests** panel, and the access name is used by other functions to access the CHD.

See the table in 14.1 for the features that apply during a call to this function. In particular, the names of cells saved in the CHD reflect any aliasing that was in force at the time the CHD was created.

The file is opened from the library search path, if a full path is not provided. The CHD is a data structure that provides information about the hierarchy in compact form, and does not use that main database. The second argument is an integer that determines the level of statistical information about the hierarchy saved. This info is available from the `ChdInfo` function and by other means. The values can be:

0	No information is saved.
1	Only total object counts are saved (default).
2	Object totals are saved per layer.
3	Object totals are saved per cell.
4	Objects counts are saved per cell and per layer.

The larger the value, the more memory is required, so it is best to only save information that will be used.

If the `ChdEstFlatMemoryUse` function will be called from the new CHD, the per-cell totals *must* be specified (value 3 or 4) or the estimate will be wildly inaccurate.

The CHD refers to physical information only. On error, a null string is returned, and an error message may be available with the `GetError` function.

(int) `WriteCellHierDigest(`*chd_name*`,` *filename*`,` *incl_geom*`,` *no_compr*`)`

This function will write a disk file representation of the Cell Hierarchy Digest (CHD) associated with the access name given as the first argument, into the file whose name is given as the second argument. Subsequently, the file can be read with `ReadCellHierDigest` to recreate the CHD. The file has no other use and the format is not documented.

The CHD (and thus the file) contains offsets onto the target archive, as well as the archive location. There is no checksum or other protection currently, so it is up to the user to make sure that the target archive is not moved or modified while the CHD is potentially or actually in use.

If the boolean argument *incl_geom* is true, and the CHD has a linked CGD (as from `ChdLinkCgd`), then geometry records will be written to the file as well. When the file is read, a new CGD will be created and linked to the new CHD. Presently, the linked CGD must have memory or file type, as described for `OpenCellGeomDigest`.

The boolean argument *no_compr*, if true, will skip use of compression of the CHD records. This is unnecessary and not recommended, unless compatibility with *Xic* releases earlier than 3.2.17, which did not support compression, is needed.

The function returns 1 if the file was written successfully, 0 otherwise, with an error message likely available from `GetError`.

(string) `ReadCellHierDigest(`*filename*`,` *cgd_type*`)`

This function returns an access name to a new cell Hierarchy Digest (CHD) created from the file whose name is passed as an argument. The file must have been created with `WriteCellHierDigest`, or with the **Save** button in the **Cell Hierarchy Digests** panel.

If the file was written with geometry records included, a new Cell Geometry Digest (CGD) may also be created (with an internally generated access name), and linked to the new CHD. If the integer argument *cgd_type* is 0, a "memory" CGD will be created, which has the compressed geometry data stored in memory. If *cgd_type* is 1, a "file" CGD will be created, which will use offsets to obtain geometry from the CHD file when needed. If *cgd_type* is any other value, or the file does not contain geometry records, no CGD will be produced.

On error, a null string is returned, with an error message probably available from `GetError`.

(stringlist_handle) `ChdList()`

This function returns a handle to a list of access strings to Cell Hierarchy Digests that are currently in memory. The function never fails, though the handle may reference an empty list.

(int) `ChdChangeName(`*old_chd_name*`,` *new_chd_name*`)`

This function allows the user to change the access name of an existing Cell Hierarchy Digest (CHD) to a user-supplied name. The new name must not already be in use by another CHD.

The first argument is the access name of an existing CHD, the second argument is the new access name, with which the CHD will subsequently be accessed. This name can be any text string, but can not be null.

The function returns 1 on success, 0 otherwise, with an error message likely available from `GetError`.

(int) `ChdIdValid(`*chd_name*`)`

This function returns one if the string argument is an access name of a Cell Hierarchy Digest currently in memory, zero otherwise.

(int) `ChdDestroy(`*chd_name*`)`

If the string argument is an access name of a Cell Hierarchy Digest (CHD) currently in memory, the CHD will be destroyed and its memory freed. One is returned on success, zero otherwise, with an error message likely available with `GetError`.

(string) `ChdInfo(`*chd_name*`, handle_or_filename, `*flags*`)`

This function provides information about the archive file represented by the Cell Hierarchy Digest (CHD) whose access name is given as the first argument. If the second argument is a string giving the name of a file, output will go to that file. If the second argument is a handle returned from the `Open` function or similar (opened for writing), output goes to the handle stream. In either case, the return value is a null string. If the second argument is a scalar 0, the output will be in the form of a string which is returned.

The third argument is an integer or string which determines the type of information to return. If an integer, the bits are flags that control the possible data fields and printing modes. The string form is a space or comma-separated list of text tokens (from the list below, case insensitive) or hex integers. The hex numbers or equivalent values for the text tokens are or'ed together to form the flags integer.

If this argument is 0, all flags except for `allcells`, `instances`, `flags`, `instcnts`, and `instcntsp` are implied. Thus, the sometimes very lengthly cells/instances listing is skipped by default. To obtain all available information, pass `-1` or `all` as the flags value.

Keyword	**Value**	**Description**
`filename`	0x1	File name.
`filetype`	0x2	File type ("`CIF`", "`CGX`", "`GDSII`", or "`OASIS`").
`unit`	0x4	File unit in meters (e.g., GDSII M-UNIT).
`alias`	0x8	Applied cell name aliasing modes.
`reccounts`	0x10	Table of record type counts (file format dependent).
`objcounts`	0x20	Table of object counts.
`depthcnts`	0x40	Tabulate the number of cell instances at each hierarchy level.
`estsize`	0x80	Print estimated memory needed to read file into *Xic*.
`estchdsize`	0x100	Print size of data structure used to provide info.
`layers`	0x200	List of layer names found, as for `ChdLayers` function.
`unresolved`	0x400	List any cells that are referenced but not defined in the file.
`topcells`	0x800	Top-level cells.
`allcells`	0x1000	All cells.
`offsort`	0x2000	Sort cells by offset in archive file.
`offset`	0x4000	Print offsets of cell definitions in archive file.
`instances`	0x8000	List instances with cells.
`bbs`	0x10000	List bounding boxes with cells, and attributes with instances.
`flags`	0x20000	Unused.
`instcnts`	0x40000	Count cell instances and report totals.
`instcntsp`	0x80000	Count cell instances and report totals per master.
`all`	-1	Set all flags.

The information provided by these flags is more fully described below.

`filename`

Print the name of the archive file for which the information applies.

`filetype`

Print a string giving the format of the archive file: one of "`CIF`", "`CGX`", "`GDSII`", or "`OASIS`".

`unit`
This is a file parameter giving the value of one unit in meters. In GDSII files, this is obtained from the M-UNIT record. The value is typically 1e-9, which means that a coordinate value of 1000 corresponds to one micron.

`alias`
Print a string giving the cell name aliasing modes that were in effect when the CHD was created.

`reccounts`
Print a table of the counts for record types found in the archive. This is format-dependent.

`objcounts`
Print a table of object counts found in the archive file. The table contains the following keywords, each followed by a number.

Keyword	**Description**
`Records`	Total record count
`Cells`	Number of cell definitions
`Boxes`	Number of rectangles
`Polygons`	Number of polygons
`Wires`	Number of wire paths
`Avg Verts`	Average vertex count per poly or wire
`Labels`	Number of (non-physical) labels
`Srefs`	Number of non-arrayed instances
`Arefs`	Number of arrayed instances

If the per-layer counts option was set when the CHD was created, additional lines will display the object counts as above, broken out per-layer.

`depthcnts`
A table of the number of cell instantiations at each hierarchy level is printed, for each top-level cell found in the file. The count for depth 0 is 1 (the top-level cell), the count at depth 1 is the number of subcells of the top-level cell, depth 2 is the number of subcells of these subcells, etc. Arrays are expanded, with each element counting as an instance placement. A total is printed, the same value that would be obtained from the `instcnts` flag.

`estsize`
This flag will enable printing of the estimated memory required to read the entire file into *Xic*. The system must be able to provide at least this much memory for a read to succeed.

`estchdsize`
Print an estimate of the memory required by the present CHD.

By default, a compression mechanism is used to reduce the data storage needed for instance lists. The NoCompressContext variable, if set, will turn off use of compression. If compression is used, the `extcxsize` field will include compression statistics. The “ratio” is the space actually used to the space used if not compressed.

`layers`
Print a list of the layer names encountered in the archive, as for the `ChdLayers` function.

`unresolved`
This will list cells that are referenced but not defined in the file. These will also be listed if `allcells` is given. A valid archive file will not contain unresolved references.

`topcells`
List the top-level cells, i.e., the cells in the file that are not used as a subcell by another cell in the file. If `allcells` is also given, only the names are listed, otherwise the cells are listed including the `offset`, `instances`, `bbs`, and `flags` fields if these flags are set. The list will be sorted as per `offsort`.

`allcells`
All cells found in the file are listed by name, including the `offset`, `instances`, `bbs`, and `flags` fields if these flags are also given. The list will be sorted as per `offsort`.

The following flags apply only if at least one of `topcells` or `allcells` is given.

`offsort`
If this flag is set, the cells will be listed in ascending order of the file offset, i.e., in the order in which the cell definitions appear in the archive file. If not set, cells are listed alphabetically.

`offset`
When set, the cell name is followed by the offset of the cell definition record in the archive file. This is given as a decimal number enclosed in square brackets.

`instances`
For each cell, the subcells used in the cell are listed. The subcell names are indented and listed below the cell name.

`bbs`
For each cell the bounding box is shown, in L,B R,T form. For subcells, the position, transformation, and array parameters are shown. Coordinates are given in microns. The subcell transformation and array parameters are represented by a concatenation of the following tokens, which follow the subcell reference position. These are similar to the transformation tokens found in CIF, and have the same meanings.

`MY`	Mirror about the x-axis.
`R`*i*,*j*	Rotate by an angle given by the vector *i*,*j*.
`M`*mag*	Magnify by *mag*.
`A`*nx*,*ny*,*dx*,*dy*	Specifies an array, *nx* x *ny* with spacings *dx*, *dy*.

Note: for technical reasons, the cell bounding boxes in CHDs do *not* include empty cells, unlike the bounding boxes computed in the main database, which will include the placement location points.

`flags`
This is currently unused and ignored.

`instcnts`
Print the total number of cell instantiations found in the hierarchy. Arrays are expanded, i.e., each element of an array counts as an instance placement.

`instcntsp`
Similar to `instcnts`, but print the total instantiations for each master cell.

`all`
This enables all flags.

On error, a null string is returned, with an error message likely available from `GetError`.

This function is similar to the **!fileinfo** command and to the `FileInfo` script function.

(string) `ChdFileName(`*chd_name*`)`
This function returns a string containing the full pathname of the file associated with the Cell Hierarchy Digest (CHD) whose access name was given in the argument. A null string is returned on error, with an error message likely available from `GetError`.

(string) `ChdFileType(`*chd_name*`)`
This function returns a string containing the file format of the archive file associated with the Cell Hierarchy Digest (CHD) whose access name was given in the argument. A null string is returned on error, with an error message likely available from `GetError`. Other possible returns are “`CIF`”, “`GDSII`”, “`CGX`”, and “`OASIS`”.

(stringlist_handle) `ChdTopCells(`*chd_name*`)`
This function returns a handle to a list of strings that contain the top-level cell names in the Cell Hierarchy Digest (CHD) whose access name was given in the argument (physical cells only). The top-level cells are those not used as a subcell by another cell in the CHD. A scalar zero is returned on error, with an error message likely available from `GetError`.

(stringlist_handle) `ChdListCells(`*chd_name*`,` *cellname*`,` *mode*`,` *all*`)`
This function returns a handle to a list of cellnames from among those found in the CHD, whose access name is given as the first argument. There are two basic modes, depending on whether the boolean argument *all* is true or not.

If *all* is true, the *cellname* argument is ignored, and the list will consist of all cells found in the CHD. If the integer *mode* argument is 0, all physical cell names are listed. If *mode* is 1, all electrical cell names will be returned. If any other value, the listing will contain all physical and electrical cell names, with no duplicates.

If *all* is false, the listing will contain the names of all cells under the hierarchy of the cell named in the *cellname* argument (including *cellname*). If *cellname* is 0, empty, or null, the default cell for the CHD is assumed, i.e., the cell which has been configured, or the first top-level cell found. The *mode* argument is 0 for physical cells, nonzero for electrical cells (there is no merging of lists in this case).

On error, a scalar 0 is returned, and a message may be available from `GetError`.

(stringlist_handle) `ChdLayers(`*chd_name*`)`
This function returns a handle to a list of strings that contain the names of layers used in the file represented by the Cell Hierarchy Digest whose access name is passed as the argument (physical cells only). For file formats that use a layer/datatype, the names are four-byte hex integers, where the left two bytes are the zero-padded hex value of the layer number, and the right two bytes are the zero-padded value of the datatype number. This applies for GDSII/OASIS files that follow the standard convention that layer and datatype numbers are 0–255. If either number is larger than 255, the layer "name" will consist of eight hex bytes, the left four for layer number, the right four for datatype.

The layers listing is available only if the CHD was created with info available, i.e., `OpenCellHierDigest` was called with the *info_saved* argument set to a value other than 0.

Each unique combination or layer name is listed. A scalar zero is returned on error, in which case an error message may be available from `GetError`.

(int) `ChdInfoMode(`*chd_name*`)`
This function returns the saved info mode of the Cell Hierarchy Digest whose access name is passed as the argument. This is the *info_saved* value passed to `OpenCellHierDigest`. The values are:

0	no information is saved.
1	only total object counts are saved.
2	object totals are saved per layer.
3	object totals are saved per cell.
4	objects counts are saved per cell and per layer.

If the CHD name is not resolved, the return value is -1, with an error message available from `GetError`.

(stringlist_handle) `ChdInfoLayers(`*chd_name*`,` *cellname*`)`
This is identical to the `ChdLayers` function when the *cellname* is 0, null, or empty. If the CHD was created with `OpenCellHierDigest` with the *info_saved* argument set to 4 (per-cell and per-layer info saved), then a *cellname* string can be passed. In this case, the return is a handle to a list of

layers used in the named cell. A scalar 0 is returned on error, with an error message probably available from `GetError`.

(stringlist_handle) `ChdInfoCells(`*chd_name*`)`

If the CHD whose access name is given as the argument was created with `OpenCellHierDigest` with the *info_saved* argument set to 3 (per-cell data saved) or 4 (per-cell and per-layer data saved), then this function will return a handle to a list of cell names from the source file. On error, a scalar 0 is returned, with an error message probably available from `GetError`.

(int) `ChdInfoCounts(`*chd_name*`,` *cellname*`,` *layername*`,` *array*`)`

This function will return object count statistics in the *array*, which must have size 4 or larger. The counts are obtained when the CHD, whose access name is given as the first argument, was created. The types of counts available depend on the *info_saved* value passed to `OpenCellHierDigest` when the CHD was created.

The array is filled in as follows:

array[0]	Box count.
array[1]	Polygon count.
array[2]	Wire count.
array[3]	Vertex count (polygons plus wires).

The following counts are available for the various *info_saved* modes.

info_saved = 0
No information is available.

info_saved = 1
Both *cellname* and *layername* arguments are ignored, the return provides file totals.

info_saved = 2
The *cellname* argument is ignored. If *layername* is 0, null, or empty, the return provides file totals. Otherwise, the return provides totals for *layername*, if found.

info_saved = 3
The *layername* argument is ignored. If *cellname* is 0, null, or empty, the return represents file totals. Otherwise, the return provides totals for *cellname*, if found.

info_saved = 4
If both arguments are 0, null, or empty, the return represents file totals. If *cellname* is 0, null, or empty, the return represents totals for the layer given. If *layername* is 0, null, or empty, the return provides totals for the cell name given. If both names are given, the return provides totals for the given layer in the given cell.

If a cell or layer is not found, or data are not available for some reason, or an error occurs, the return value is 0, and an error message may be available from `GetError`. Otherwise, the return value is 1, and the array is filled in.

(int) `ChdCellBB(`*chd_name*`,` *cellname*`,` *array*`)`

This returns the bounding box of the named cell. The *cellname* is a string giving the name of a physical cell found in the Cell Hierarchy Digest (CHD) whose access name is given in the first argument.

The *cellname*, if nonzero, must be the cell name after any aliasing that was in force when the CHD was created. If *cellname* is passed 0, the default cell for the CHD is understood. This is a cell name configured into the CHD, or the first top-level cell found in the archive file.

The values are returned in the *array*, which must have size 4 or larger. the order is l,b,r,t. One is returned on success, zero otherwise, with an error message likely available from `GetError`.

The cell bounding boxes for geometry are computed as the file is read, so that if the NoReadLabels variable is set during the read, i.e., when `OpenCellHierDigest` is called, text labels will not contribute to the bounding box computation.

(int) `ChdSetDefCellName(`*chd_name*`,` *cellname*`)`
This will set or unset the configuration of a default cell namein the Cell Hierarchy Digest whose access name is given in the first argument.

If the *cellname* argument in not 0 or null, it must be a cell name after any aliasing that was in force when the CHD was created, that exists in the CHD. This will set the default cell name for the CHD which will be used subsequently by the CHD whenever a cell name is not otherwise specified. The current default cell name is returned from the `ChdDefCellName` function. If *cellname* is 0 or null, the default cell name is unconfigured. In this case, the CHD will use the first top-level cell found (lowest offset on the archive file). A top-level cell is one that is not used as a subcell by any other cell in the CHD.

One is returned on success, zero otherwise, with an error message likely available with `GetError`.

(string) `ChdDefCellName(`*chd_name*`)`
This will return the default cell name of the Cell Hierarchy Digest whose access name is given in the argument. This will be the cell name configured (with `ChdSetDefCellName`), or if no cell name is configured the return will be the name of the first top-level cell found (lowest offset on the archive file). A top-level cell is one that is not used as a subcell by any other cell in the CHD.

On error, a null string is returned, with an error message likely available from `GetError`.

(int) `ChdLoadGeometry(`*chd_name*`)`
This function will read the geometry from the original layout file from the Cell Hierarchy Digest (CHD) whose access name is given in the argument into a new Cell Geometry Digest (CGD) in memory, and configures the CHD to link to the new CGD for use when reading. The new CGD is given an internally-generated access name, and will store all geometry data in memory. The new CGD will be destroyed when unlinked.

This is a convenience function, one can explicitly create a CGD (with `OpenCellGeomDigest`) and link it to the CHD (with `ChdLinkCgd`) if extended features are needed.

See the table in 14.1 for the features that apply during a call to this function.

The return value is 1 on success, 0 otherwise, with an error message likely available from `GetError`.

(int) `ChdLinkCgd(`*chd_name*`,` *cgd_name*`)`
This function links or unlinks a Cell Geometry Digest (CGD) whose access name is given as the second argument, to the Cell Hierarchy Digest (CHD) whose access name is given as the first argument. With a CGD linked, when the CHD is used to access geometry data, the data will be obtained from the CGD, if it exists in the CGD, and from the original layout file if not provided by the CGD. The CGD is a "geometry cache" which resides in memory.

If the *cgd_name* is null or empty (0 can be passed for this argument) any CGD linked to the CHD will be unlinked. If the CGD was created specifically to link with the CHD, such as with `ChdLoadGeometry`, it will be freed from memory, otherwise it will be retained.

This function returns 1 on success, 0 otherwise with an error message likely available from `GetError`.

(string) `ChdGetGeomName(`*chd_name*`)`
The string argument is an access name for a Cell Hierarchy Digest (CHD) in memory. If the CHD exists and has an associated Cell Geometry Digest (CGD) linked (e.g., `ChdLoadGeometry` was called), this function returns the access name of the CGD. If the CHD is not found or not configured with a CGD, a null string is returned.

(int) `ChdClearGeometry(`*chd_name*`)`

This function will clear the link to the Cell Geometry Digest within the Cell Hierarchy Digest. If a CGD was linked, and it was created explicitly for linking into the CHD as in `ChdLoadGeometry`, the CGD will be freed, otherwise it will be retained. The return value is 1 if the CHD was found, 0 otherwise, with a message available from `GetError`.

This function is identical to `ChdLinkCgd` with a null second argument.

(int) `ChdSetSkipFlag(`*chd_name*`,` *cellname*`,` *skip*`)`

This will set/unset the skip flag in the Cell Hierarchy Digest (CHD) whose access name is given in the first argument for the cell named in *cellname* (physical only).

The *cellname*, if nonzero, must be the cell name after any aliasing that was in force when the CHD was created. If *cellname* is passed 0, the default cell for the CHD is understood. This is a cell name configured into the CHD, or the first top-level cell found in the archive file.

With the skip flag set, the cell is ignored in the CHD, i.e., the cell and its instances will not be included in output or when reading into memory when the CHD is used to access layout data. The last argument is a boolean value: 0 to unset the skip flag, nonzero to set it. The return value is 1 if a flag was altered, 0 otherwise, with an error message likely available from `GetError`.

(int) `ChdClearSkipFlags(`*chd_name*`)`

This will clear the skip flags for all cells in the Cell Hierarchy Digest whose access name is given in the argument. The skip flags are set with `SetSkipFlag`. The return value is 1 on success, 0 otherwise, with an error message likely available with `GetError`.

(int) `ChdCompare(`*chd_name1*`,` *cname1*`,` `chd_name2`, *cname2*`,` *layer_list*`,` *skip_layers*`,` *maxdiffs*`,` *obj_types*`,` *geometric*`,` *array*`)`

This will compare the contents of two cells, somewhat similar to the **!compare** command and the **Compare Layouts** operation in the **Convert Menu**. However, only one cell pair is compared, taking account only of features within the cells. The `ChdCompareFlat` function is similar, but flattens geometry before comparison.

When comparing subcells, arrays will be expanded into individual instances before comparison, avoiding false differences between arrayed and unarrayed equivalents. The returned handles (if any) contain differences, as lists of object copies. Properties are ignored.

The arguments are:

chd_name1
: Access name of a Cell Hierarchy Digest (CHD) in memory.

cname1
: Name of cell in *chd_name1* to compare, if null (0 passed) the default cell in *chd_name1* is used.

chd_name2
: If not null or empty (one can pass 0 for this argument), the name of another CHD.

cname2
: Name of cell in the second CHD, or in memory, to compare. If null, or 0 is passed, and a second CHD was specified, the second CHD's default cell is understood. Otherwise, the name will be assumed the same as *cname1*.

layer_list
: String of space-separated layer names, or zero which implies all layers.

skip_layers
: If this boolean value is nonzero and a *layer_list* was given, the layers in the list will be skipped. Otherwise, only the layers in the list will be compared (all layers if *layer_list* is passed zero).

maxdiffs
The function will return after recording this many differences. If 0 or negative, there is no limit.

obj_types
String consisting of the layers `c,b,p,w,l`, which determines objects to consider (subcells, boxes, polygons, wires, and labels), or zero. If zero, "`cbpw`" is the default, i.e., labels are ignored. If the geometric argument is nonzero, all but '`c`' will be ignored, and boxes, polygons, and wires will be compared.

geometric
If this boolean value is nonzero, a geometric comparison will be performed, otherwise objects are compared directly.

array
This is a two-element or larger array, or zero. If an array is passed, upon return the elements are handles to lists of box, polygon, and wire object copies (labels and subcells are not returned): *array*[0] contains a list of objects in handle1 and not in handle2, and *array*[1] contains objects in handle2 and not in handle1. The `H` function must be used on the array elements to access the handles. If the argument is passed zero, no object lists are returned.

The cells for the current mode (electrical or physical) are compared. The scalar return can take the following values:

-1	An error occurred, with a message possibly available from the `GetError` function.
0	Successful comparison, no differences found.
1	Successful comparison, differences found.
2	The cell was not found in *chd_name1*.
3	The cell was not found in *chd_name2*.
4	The cell was not found in either source.

(int) `ChdCompareFlat(`*chd_name1*`,` *cname1*`,` `chd_name2,` *cname2*`,` *layer_list*`,` *skip_layers*`,` *maxdiffs*`,` *area*`,` *coarse_mult*`,` *find_grid*`,` *array*`)`
This will compare the contents of two hierarchies, using a flat geometry model similar to the flat options of the **!compare** command and the **Compare Layouts** operation in the **Convert Menu**. The `ChdCompare` function is similar, but does not flatten.

The returned handles (if any) contain the differences, as lists of objects. Properties are ignored.

The arguments are:

chd_name1
Access name of a Cell Hierarchy Digest (CHD) in memory.

cname1
Name of cell in *chd_name1* to compare, if null (0 passed) the default cell in *chd_name1* is used.

chd_name2
Access name of another CHD in memory. This argument can not be null as in `ChdCompare`, flat comparison to memory cells is unavailable.

cname2
Name of cell in the second CHD to compare. If null, or 0 is passed, the second CHD's default cell is understood.

layer_list
String of space-separated layer names, or zero which implies all layers.

skip_layers
If this boolean value is nonzero and a *layer_list* was given, the layers in the list will be skipped. Otherwise, only the layers in the list will be compared (all layers if *layer_list* is passed zero).

maxdiffs
The function will return after recording this many differences. If 0 or negative, there is no limit.

area
This argument can be an array of size 4 or larger, or 0. If an array, it contains a rectangle description in order L,B,R,T in microns, which specifies the area to compare. If 0 is passed, the area compared will contain the two hierarchies entirely.

coarse_mult
The comparison is performed in the manner described for the `ChdIterateOverRegion` function, using a fine grid and a coarse grid. This argument specifies the size of the coarse grid in multiples of the fine grid size. All of the geometry needed for a coarse grid cell is brought into memory at once, so this size should be consistent with memory availability and layout feature density. Values of 1–100 are accepted for this argument, with 20 a reasonable initial choice.

fine_grid
Comparison is made within a fine grid cell. The optimum fine grid size depends on factors including layout feature density and memory availability. Larger sizes usually run faster, but may require excessive memory. The value is given in microns, with the acceptable range being 1.0 – 100.0 microns. A reasonable initial choice is 20.0, but experimentation can often yield better performance.

array
This is a two-element or larger array, or zero. If an array is passed, upon return the elements are handles to lists of box, polygon, and wire object copies (labels and subcells are not returned): *array*[0] contains a list of objects in handle1 and not in handle2, and *array*[1] contains objects in handle2 and not in handle1. The `H` function must be used on the array elements to access the handles. If the argument is passed zero, no object lists are returned.

The cells for the physical mode are compared, it is not possible to compare electrical cells in flat mode. The return value is an integer, -1 on error (with a message likely available from `GetError`), 0 if no differences were seen, or positive giving the number of differences seen.

(int) `ChdEdit(`*chd_name*`,` *scale*`,` *cellname*`)`
This will read the given cell and its descendents into memory and open the cell for editing, similar to the `Edit` function, however the layout data will be accessed through the Cell Hierarchy Digest whose access name is given in the first argument. The return value takes the same values as the `Edit` function return.

See the table in 14.1 for the features that apply during a call to this function.

The *scale* will multiply all coordinates in cells opened, and can be in the range 0.001 – 1000.0.

The *cellname*, if nonzero, must be the cell name after any aliasing that was in force when the CHD was created. If *cellname* is passed 0, the default cell for the CHD is understood. This is a cell name configured into the CHD, or the first top-level cell found in the archive file.

(int) `ChdOpenFlat(`*chd_name*`,` *scale*`,` *cellname*`,` *array*`,` *clip*`)`
This will read the cell named in the *cellname* string and its subcells into memory, creating a flat cell with the same name. The Cell Hierarchy Digest (CHD) whose access name is given in the first argument is used to obtain the layout data.

See the table in 14.1 for the features that apply during a call to this function. Text labels are ignored.

The *cellname*, if nonzero, must be the cell name after any aliasing that was in force when the CHD was created. If *cellname* is passed 0, the default cell for the CHD is understood. This is a cell name configured into the CHD, or the first top-level cell found in the archive file.

If the cell already exists in memory, it will be overwritten.

The *scale* will multiply all coordinates read, and can be in the range 0.001 – 1000.0.

If the *array* argument is passed 0, no windowing will be used. Otherwise the array should have four components which specify a rectangle, in microns, in the coordinates of *cellname*. The values are

array`[0]`	X left
array`[1]`	Y bottom
array`[2]`	X right
array`[3]`	Y top

If an array is given, only the objects and subcells needed to render the window will be read.

If the boolean value *clip* is nonzero and an array is given, objects will be clipped to the window. Otherwise no clipping is done.

Before calling `ChdOpenFlat`, the memory use can be estimated by calling the `ChdEstFlatMemoryUse` function. An overall transformation can be set with `ChdSetFlatReadTransform`, in which case the area given applies in the "root" coordinates.

The return value is 1 on success, 0 on error, or -1 if an interrupt was received. In the case of an error return, an error message may be available through `GetError`.

(real) `ChdSetFlatReadTransform(`*tfstring*`,` *x*`,` *y*`)`

This rather arcane function will set up a transformation which will be used during calls to the following functions:

```
ChdOpenFlat
ChdWriteSplit
ChdGetZlist
ChdOpenOdb
ChdOpenZdb
ChdOpenZbdb
```

The transform will be applied to all of the objects read through the CHD with these functions. Why might this function be used? Consider the following: suppose we have a CHD describing a cell hierarchy, the top-level cell of which is to be instantiated under another cell we'll call "root", with a given transformation. We would like to consider the objects from the CHD from the perspective of the "root" cell. This function would be called to set the transformation, then one of the flat read functions would be called and the returned objects accumulated. The returned objects will have coordinates relative to the "root" cell, rather than relative to the top-level cell of the CHD.

The *tfstring* describes the rotation and mirroring part of the transformation. It is either one of the special tokens to be described, or a sequence of the following tokens:

`MX`
: Flip the X axis.

`MY`
: Flip the Y axis.

`R`*nnn*
: Rotate by *nnn* degrees. The *nnn* must be one of 0, 45, 90, 135, 180, 225, 270, 315.

White space can appear between tokens. The operations are performed in order. Note that, e.g., "`MXR90`" is very different from "`R90MX`".

Alternatively, the *tfstring* can contain a single "Lef/Def" token as listed below. The second column is the equivalent string using the syntax previously described.

`N`	null or empty or `R0`
`S`	`R180`
`W`	`R90`
`E`	`R270`
`FN`	`MX`
`FS`	`MY`
`FW`	`MYR90`
`FE`	`MXR90`

The x and y are the translation part of the transformation. These are coordinates, given in microns.

If *tfstring* is null or empty, no rotations or mirroring will be used.

The function returns 1 on success, 0 if the *tfstring* contains an error.

(real) `ChdEstFlatMemoryUse(`*chd_name*`, cellname,` *array*`,` *counts_array*`)`

This function will return an estimate of the memory required to perform a `ChdOpenFlat` call. The first argument is the access name of an existing Cell Hierarchy Digest that was created with per-cell object counts saved (e.g., a call to `OpenCellHierDigest` with the *info_saved* argument set to 3 or 4).

The *cellname*, if nonzero, must be the cell name after any aliasing that was in force when the CHD was created. If *cellname* is passed 0, the default cell for the CHD is understood. This is a cell name configured into the CHD, or the first top-level cell found in the archive file.

The third argument is an array of size four or larger that contains the rectangular area as passed to the `ChdOpenFlat` call. The components are

array[0]	X left
array[1]	Y bottom
array[2]	X right
array[3]	Y top

This argument can also be zero to indicate that the full area of the top level cell is to be considered.

The final argument is also an array of size four or larger, or zero. If an array is passed, and the function succeeds, the components are filled with the following values:

counts_array[0]	estimated total box count
counts_array[1]	estimated total polygon count
counts_array[2]	estimated total wire count
counts_array[3]	estimated total vertex count

These are counts of objects that would be saved in the top-level cell during the `ChdOpenFlat` call. These are estimates, based on area normalization, and do not include any clipping or merging. The vertex count is an estimate of the total number of polygon and wire vertices.

The return value is an estimate, in megabytes, of the incremental memory required to perform the `ChdOpenFlat` call. This does not include normal overhead.

(int) `ChdWrite(`*chd_name*`,` *scale*`,` *cellname*`,` *array*`,` *clip*`,` *all*`,` *flatten*`,` *ecf_level*`,` *outfile*`)`

This will write the cell named in the *cellname* string to the output file given in *outfile*, using the Cell Hierarchy Digest whose access name is given in the first argument to obtain layout data.

If the *outfile* is null or empty, the geometry will be "written" as cells in the main database, hierarchically if *all* is true. This allows windowing to be applied when converting a hierarchy, which will attempt to convert only objects and cells needed to render the window area. This has the potential to hopelessly scramble your in-memory design data so be careful.

See the table in 14.1 for the features that apply during a call to this function.

The *cellname*, if nonzero, must be the cell name after any aliasing that was in force when the CHD was created. If *cellname* is passed 0, the default cell for the CHD is understood. This is a cell name configured into the CHD, or the first top-level cell found in the archive file.

If the boolean argument *all* is nonzero, the hierarchy under the cell is written, otherwise only the named cell is written. If the *outfile* is null or empty, native cell files will be created in the current directory. If the *outfile* is the name of an existing directory, native cell files will be created in that directory. Otherwise, the extension of the *outfile* determines the file type:

CGX	`.cgx`
CIF	`.cif`
GDSII	`.gds, .str, .strm, .stream`
OASIS	`.oas`

Only these extensions are recognized, however CGX and GDSII allow an additional `.gz` which will imply compression.

The *scale* will multiply all coordinates read, and can be in the range 0.001 – 1000.0.

If the *array* argument is passed 0, no windowing will be used. Otherwise the array should have four components which specify a rectangle, in microns, in the coordinates of *cellname*. The values are

array`[0]`	X left
array`[1]`	Y bottom
array`[2]`	X right
array`[3]`	Y top

If an array is given, only the objects and subcells needed to render the window will be written.

If the boolean value *clip* is nonzero and an array is given, objects will be clipped to the window. Otherwise no clipping is done.

If the boolean value *all* is nonzero, the hierarchy under *cellname* is written, otherwise not. If windowing is applied, this applies only to *cellname*, and not subcells.

If the boolean variable *flatten* is nonzero, the objects in the hierarchy under *cellname* will be written into *cellname*, i.e., flattened. The *all* argument is ignored in this case. Otherwise, no flattening is done.

The *ecf_level* is an integer 0–3 which sets the empty cell filtering level, as described for the **Format Conversion** panel in 14.10. The values are

0	No empty cell filtering.
1	Apply pre- and post-filtering.
2	Apply pre-filtering only.
3	Apply post-filtering only.

The return value is 1 on success, 0 on error, or -1 if an interrupt was received. In the case of an error return, an error message may be available through `GetError`.

(int) `ChdWriteSplit(`*chd_name*`,` *cellname*`,` *basename*`,` *array*`,` *regions_or_gridsize*`,` *numregions_or_bloatval*`,` *maxdepth*`,` *scale*`,` *flags*`)`
This function will read the geometry data through the a Cell Hierarchy Digest (CHD) whose name

is given as the first argument, into a collection of files representing rectangular regions of the top-level cell. Each output file contains only the cells and geometry necessary to represent the region. The regions can be specified as a list of rectangles, or as a grid.

See the table in 14.1 for the features that apply during a call to this function.

cellname
: The *cellname*, if nonzero, must be the cell name after any aliasing that was in force when the CHD was created. If *cellname* is passed 0, the default cell for the CHD is understood. This is a cell name configured into the CHD, or the first top-level cell found in the archive file.

basename
: The *basename* is a cell path name in the form

  [*/path/to/*]*basename.ext*,

  where the extension *ext* gives the type of file to create. One of the following extensions must be provided:

CGX output	`.cgx`
CIF output	`.cif`
GDSII output	`.gds, .str, .strm, .stream`
OASIS output	`.oas, .oasis`

  A "`.gz`" second extension is allowed following CGX and GDSII extensions in which case the files will be compressed using the `gzip` format.

  When writing a list of regions, the output files will be named in the form *basename_N.ext*, where the *.ext* is the extension supplied, and N is a 0–based index of the region, ordered as given. When writing a grid, the output files will be named in the form *basename_X_Y.ext*, where the *.ext* is the extension supplied, and X, Y are integer indices representing the grid cell (origin is the lower-left corner). If a directory path is prepended to the *basename*, the files will be found in that directory (which must exist, it will not be created).

array
: The *array* argument can be 0, or the name of an array of size four or larger that contains a rectangle specification, in microns, in order L,B,R,T. If given, the rectangle should intersect the bounding box of the top-level cell (*cellname*). Only cells and geometry within this area will be written to output. If 0 is passed, the entire bounding box of the top cell is understood.

  When writing grid files, the origin of the grid, before bloating, is at the lower-left corner of the area to be output.

regions_or_gridsize
: This argument can be an array, or a scalar value. If an array, the array consists of one or more rectangular area specifications, in order L,B,R,T in microns. These are the regions that will be written to output files.

  If this argument is a number, it represents the size of a square grid cell, in microns.

bloatval
: If an array was passed as the previous argument, then this argument is an integer giving the number of regions in the array to be written. The size of the array is at least four times the number of regions.

  If instead a grid value was given in the previous argument, then this argument provides a bloating value. The grid cells will be bloated by this value (in microns) if the value is nonzero. A positive value pushes out the grid cell edges by the value given, a negative value does the reverse.

maxdepth
: This integer value applies only when flattening, and sets the maximum hierarchy depth for include in output. If 0, only objects in the top-level cell will be included,

scale

This is a scale factor which will be applied to all output. The *gridsize*, *bloatval*, and *array* coordinates are the sizes found in output, and are independent of the scale factor. The valid range is 0.001 – 1000.0.

flags

This argument is a **string** consisting of specific letters, the presence of which sets one of several available modes. These are

`p`	parallel
`f`	flatten
`c`	clip
`n`[*N*]	empty cell filtering
`m`	map names

The character recognition is case-insensitive. A null or empty string indicates no flags set.

`p`

If `p` is given, a parallel writing algorithm is used. Otherwise, the output files are generated in sequence. The files should be identical from either writing mode. The parallel mode may be a little faster, but requires more internal memory. When writing in parallel, the user may encounter system limitations on the number of file descriptors open simultaneously.

`f`

If `f` is given, the output will be flattened. When flattening, an overall transformation can be set with `ChdSetFlatReadTransform`, in which case the given area description would apply in the "root" coordinates.

If not given, the output files will be hierarchical, but only the subcells needed to render the grid cell area, each containing only the geometry needed, will be written.

`c`

If `c` is given, objects will be clipped at the grid cell boundaries. This also applies to objects in subcells, when not flattening.

`n`[*N*]

The '`n`' can optionally be followed by an integer 0–3. If no integer follows, '3' is understood. This sets the empty cell filtering level as described for the **Format Conversion** panel in 14.10. The values are

0 No empty cell filtering (no operation).
1 Apply pre- and post-filtering.
2 Apply pre-filtering only.
3 Apply post-filtering only.

`m`

If `m` is given, and `f` is also given (flattening), the top-level cell names in the output files will be modified so as to be unique in the collection. A suffix "_*N*" is added to the cell name, where *N* is a grid cell or region index. The index is 0 for the lower-left grid cell, and is incremented in the sweep order left to right, bottom to top. If writing regions, the index is 0–based, in the order of the regions given. Furthermore, a native cell file is written, named "*basename*_root", which calls each of the output files. Loading this file will load the entire output collection, memory limits permitting.

The function returns 1 on success, 0 otherwise, with an error message likely available from `GetError`.

(int) `ChdCreateReferenceCell(`*chd_name*`,` *cellname*`)`

This function will create a reference cell (see 8.9.3) in memory. A reference cell is a special cell that references a cell hierarchy in an archive file, but does not have its own content. Reference cells can be instantiated during editing like any other cell, but their content is not visible. When a reference cell is written to disk as part of a cell hierarchy, the hierarchy of the reference cell is extracted from its source and streamed into the output.

The first argument is a string giving the name of a Cell Hierarchy Digest (CHD) already in memory. The second argument is the name of a cell in the CHD, which must include aliasing if aliasing was applied when the CHD was created. This will also be the name of the reference cell. A cell with this name should not already exist in current symbol table.

Although the CHD is required for reference cell creation, it is not required when the reference cell is written, but will be used if present. The archive file associated with the CHD should not be moved or altered before the reference cell is written to disk.

A value 0 is returned on error, with a message probably available from `GetError`. The value 1 is returned on success.

(int) `ChdLoadCell(`*chd_name*`,` *cellname*`)`

This function will load a cell into the main editing database, and subcells of the cell will be loaded as reference cells (see 8.9.3). This allows the cell to be edited, without loading the hierarchy into memory. When written to disk as part of a hierarchy, the cell hierarchies of the reference cells will be extracted from the input source and streamed to output.

The first argument is a string giving the name of a Cell Hierarchy Digest (CHD) already in memory. The second argument is the name of a cell in the CHD, which must include aliasing if aliasing was applied when the CHD was created. This cell will be read into memory. Any subcells used by the cell will be created in memory as reference cells, which a special cells which have no content but point to a source for their content.

Although the CHD is required for reference cell creation, it is not required when the reference cell is written, but will be used if present. The archive file associated with the CHD should not be moved or altered before the reference cell is written to disk.

A value 0 is returned on error, with a message probably available from `GetError`. The value 1 is returned on success.

(int) `ChdIterateOverRegion(`*chd_name*`,` *cellname*`,` *funcname*`,` *array*`,` *coarse_mult*`,` *fine_grid*`,` *bloat_val*`)`

This function is an interface to a system which creates a logical rectangular grid over a cell hierarchy, then iterates over the partitions in the grid, performing some action on the logically flattened geometry.

A Cell Hierarchy Digest (CHD) is used to obtain the flattened geometry, with or without the assistance of a Cell Geometry Digest (CGD). There are actually two levels of gridding: the coarse grid, and the fine grid. The area of interest is first logically partitioned into the coarse grid. For each cell of the coarse grid, a "ZBDB" special database (see F.7.4) is created, using the fine grid. For example, one might choose 400x400 microns for the coarse grid, and 20x20 microns for the fine grid. Thus, geometry access is in 400x400 "chunks". The geometry is extracted, flattened, and split into separate trapezoid lists for each fine grid area, for each layer.

As each fine grid cell is visited, a user-supplied script function is called. The operations performed are completely up to the user, and the framework is intended to be as flexible as possible. As an example, one might extract geometric parameters such as density, minimum line width and spacing, for use by a process analysis tool. Scalar parameters can be conveniently saved in spatial parameter tables (SPTs, see F.7.3).

The first argument is the access name of a CHD in memory. The second argument is the top-level cell from the CHD, or if passed 0, the CHD's default cell will be used.

The third argument is the name of a user-supplied script function which will implement the user's calculations. The function should already be in memory before `ChdIterateOverRegion` is called. This function is described in more detail below.

The *array* argument can be 0, in which case the area of interest is the entire top-level cell. Otherwise, the argument should be an array of size four or larger containing the rectangular area of interest, in order L,B,R,T in microns. The coarse and find grid origin is at the lower left corner of the area of interest.

The *fine_grid* argument is the size of the fine grid (which is square) in microns. The *coarse_mult* is an integer representing the size of the coarse grid, in *fine_grid* quanta.

The *bloat_val* argument specifies an amount, in microns, that the grid cells (both coarse and fine) should be expanded when returning geometry. Geometry is clipped to the bloated grid. Thus, it is possible to have some overlap in the geometry returned from adjacent grid cells. This value can be negative, in which case grid cells will effectively shrink.

The callback function has the following prototype.

(int) `callback`(*db_name*, *j*, *i*, *spt_x*, *spt_y*, *data*, *cell_name*, *chd_name*)

The function definition must start with the *db_name* and include the arguments in the order shown, but unused arguments to the right of the last needed argument can be omitted.

db_name (string)
The access name of the ZBDB database containing geometry.

j (integer)
The X index of the current fine grid cell.

i (integer)
The Y index of the current fine grid cell.

spt_x (real)
The X coordinate value in microns of the current grid cell in a spatial parameter table:
coarse_grid_cell_left + *j***fine_grid_size* + *fine_grid_size*/2

spt_y (real)
The Y coordinate value in microns of the current grid cell in a spatial parameter table:
coarse_grid_cell_bottom + *i***fine_grid_size* + *fine_grid_size*/2

data (real array)
An array containing miscellaneous parameters, described below).

cell_name (string)
The name of the top-level cell.

chd_name (string)
The access name of the CHD.

The *data* argument is an array that contains the following parameters.

index	**description**
0	The spatial parameter table column size.
1	The spatial parameter table row size.
2	The fine grid period in microns.
3	The coarse grid period in microns.
4	The amount of grid cell expansion in microns.
5	Area of interest left in microns.
6	Area of interest bottom in microns.
7	Area of interest right in microns.
8	Area of interest top in microns.
9	Coarse grid cell left in microns.
10	Coarse grid cell bottom in microns.
11	Coarse grid cell right in microns.
12	Coarse grid cell top in microns.
13	Fine grid cell left in microns.
14	Fine grid cell bottom in microns.
15	Fine grid cell right in microns.
16	Fine grid cell top in microns.

The trapezoid data for the grid cells can be accessed, from within the callback function, with the `GetZlistZbdb` function.

`GetZlistZbdb`(*db_name*, *layer_lname*, *j*, *i*)

Example:
Here is a function that simply prints out the fine grid indices, and the number of trapezoids in the grid location on a layer named "`M1`".

```
function myfunc(dbname, j, i, x, y, prms)
    zlist = GetZlistZbdb(dbname, "M1", j, i)
    Print("Location", j, i, "contains", Zlength(zlist), "zoids on M1")
endfunc
```

If the function returns a nonzero value, the operation will abort. If there is no explicit return statement, the return value is 0.

```
if (some error)
    return 1
end
```

If all goes well, `ChdIterateOverRegion` returns 1, otherwise 0 is returned, with an error message possibly available from `GetError`.

This function is intended for OEM users, customization is possible. Contact Whiteley Research for more information.

(int) `ChdWriteDensityMaps(`*chd_name*`,` *cellname*`,` *array*`,` *coarse_mult*`,` *fine_grid*`,` *bloat*`,` *save*`)`

This function uses the same framework as `ChdIterateOverRegion`, but is hard-coded to extract density values only. The *chd_name*, *cellname*, *array*, *coarse_mult*, and *fine_grid* arguments are as described for that function.

When called, the function will iterate over the given area, and compute the fraction of dark area for each layer in a fine grid cell, saving the values in a spatial parameter table (SPT, see F.7.3). The access names of these SPTs are in the form *cellname.layername*, where *cellname* is the name of the top-level cell being processed. The *layername* is the name of the layer, possibly in hex format as used elsewhere.

If the boolean *save* argument is nonzero, the SPTs will be retained in memory after the function returns. Otherwise, the SPTs will be dumped to files in the current directory, and destroyed. The file names are the same as the SPT names, with a "`.spt`" extension added. These files can be read with `ReadSPtable`, and are in the format described for that function, with the "reference coordinates" the central points of the fine grid cells.

If all goes well, `ChdWriteDensityMaps` returns 1, otherwise 0 is returned, with an error message possibly available from `GetError`.

F.4.11 Cell Geometry Digest

(string) `OpenCellGeomDigest(`*idname*`,` *string*`,` *type*`)`

This function returns an access name to a new Cell Geometry Digest (CGD) which is created in memory. A CGD is a data structure that provides access to cell geometry saved in compact form, and does not use the main cell database. The CGD refers to physical data only. The new CGD will be listed in the **Cell Geometry Digests** panel, and the access name is used by other functions to access the CGD.

See the table in 14.1 for the features that apply during a call to this function. In particular, the names of cells saved in the CGD reflect any aliasing that was in force at the time the CGD was created.

The first argument is a specified access name (which will be returned on success). This name can not be in use, meaning that the name can not access an existing CGD which is currently linked to a CHD. If there is a name match to an unlinked CGD, the new CGD will replace the old (which is destroyed). This argument can be passed 0 or an empty string. If a null or empty string is passed, a new access name will be generated and assigned.

The third argument is an integer 0–2 which specifies the type of CGD to create. The second (*string*) argument depends on what type of CGD is being created.

Type 0 (actually, *type* not 1 or 2)

This will create a "memory" CGD, where all geometry data will be stored in memory, in highly-compressed form. This provides the most efficient access, but very large databases may exceed memory limitations.

In this mode, the *string* argument can be one of the following:

1. A layout (archive) file. The file will be read and the geometry extracted.
2. The access name of a Cell Hierarchy Digest (CHD) in memory. The CHD will be used to read the geometry from the file it references.
3. A saved CHD file. The file will be read, and a new CHD will be created in memory. This CHD will be used to read the geometry from the file referenced.
4. A saved CGD file name. The file will be read into an in-memory CGD.

Files are opened from the library search path, if a full path is not provided.

Type 1

This will create a "file" CGD, where geometry data are stored in a CGD file on disk, and geometry is retrieved when needed via saved file offsets. This uses less memory, but is not quite as fast as saving geometry data in memory. It is generally much faster than reading geometry from the original layout file since 1) the data are highly compressed, and 2) the objects are pre-sorted by layer.

In this mode, the *string* is a path to a saved CGD file, or to a saved CHD file containing geometry records. The in-memory CGD will access this file. The file is opened from the library search path, if a full path is not provided.

Type 2

This will create a stub CGD which obtains geometry information from a remote host which is running *Xic* in server mode. The server must have a CGD in memory, from which data are obtained.

In this mode, the *string* must be in the format

hostname[:*port*]/*idname*

The [...] indicates "optional" and is not literal. The *hostname* is the network name of the machine running the server. If the server is using a non-default port number, the same port number should be provided after the host name, separated by a colon. Following the hostname or port is the access name on the server of the CGD to access, separated by a forward slash. The entire string should contain no white space.

On error, a null string is returned, and an error message may be available with the `GetError` function.

(string) `NewCellGeomDigest()`

This function creates a new, empty Cell Geometry Digest, and returns the access name. The `CgdAddCells` function can be used to add cell geometry.

(int) `WriteCellGeomDigest(`*cgd_name*`,` *filename*`)`

This function will write a disk file representation of the Cell Geometry Digest (CGD) associated with the access name given as the first argument, into the file whose name is given as the second argument. Subsequently, the file can be read with `OpenCellGeomDigest` to recreate the CGD. The file has no other use and the format is not documented.

The function returns 1 if the file was written successfully, 0 otherwise, with an error message likely available from `GetError`.

(stringlist_handle) `CgdList()`

This function returns a handle to a list of access strings to Cell Geometry Digests that are currently in memory. The function never fails, though the handle may reference an empty list.

(int) `CgdChangeName(`*old_cgd_name*`,` *new_cgd_name*`)`

This function allows the user to change the access name of an existing Cell Geometry Digest (CGD) to a user-supplied name. The new name must not already be in use by another CGD.

The first argument is the access name of an existing CGD, the second argument is the new access name, with which the CGD will subsequently be accessed. This name can be any text string, but can not be null.

The function returns 1 on success, 0 otherwise, with an error message likely available from `GetError`.

(int) `CgdIsValid(`*cgd_name*`)`

This function returns one if the string argument is an access name of a Cell Geometry Digest currently in memory, zero otherwise.

(int) `CgdDestroy(`*cgd_name*`)`

The string argument is the access name of a Cell Geometry Digest (CGD) currently in memory. If the CGD is not currently linked to a Cell Hierarchy Digest (CHD), then the CGD will be destroyed and its memory freed. One is returned on success, zero otherwise, with an error message likely available with `GetError`.

(int) `CgdIsValidCell(`*cgd_name*`,` *cellname*`)`

This function will return 1 if a Cell Geometry Digest (CGD) with an access name given as the first argument exists and contains data for the cell whose name is given as the second argument. Otherwise, 0 is returned.

(int) `CgdIsValidLayer(`*cgd_name*`,` *cellname*`,` *layername*`)`

This function returns 1 if the *cgd_name* is an access name of a Cell Geometry Digest (CGD) in memory, which contains a cell *cellname* that has data for layer *layername*. Otherwise, 0 is returned.

(int) `CgdRemoveCell(`*cgd_name*`,` *cellname*`)`

This function will remove and destroy the data for the cell *cellname* from the Cell Geometry Digest (CGD) with access name *cgd_name*. This applies to all CGD types, as described for `OpenCellGeomDigest`. If the CGD is accessing geometry from a remote server, the cell data are removed from the server.

The names of cells that have been removed are retained, and can be checked with `CgdIsCellRemoved`.

If the CGD is found and it contains *cellname*, the cell data are destroyed and the function returns 1. Otherwise, 0 is returned, with an error message available from `GetError`.

(int) `CgdIsCellRemoved(`*cgd_name*`,` *cellname*`)`

This function returns 1 if a CGD is found with access name as given in *cgd_name*, and the *cellname* is the name of a cell that has been removed from the CGD, for example with `CgdRemoveCell`. Otherwise, the return value is 0.

(int) `CgdRemoveLayer(`*cgd_name*`,` *cellname*`,` *layername*`)`

If the Cell Geometry Digest (CGD) exists, and contains data for a cell *cellname* that contains data for *layername*, the *layername* data will be deleted from the *cellname* record, and the function returns 1. Otherwise, 0 is returned, with an error message likely available from `GetError`.

This applies to memory and file type CGDs, as described for `OpenCellGeomDigest`. The data, if found, are freed, and (unlike `CgdRemoveCell`) no record of removed layers is retained. This actually reduces memory use only for memory type CGDs.

(int) `CgdAddCells(`*cgd_name*`,` *chd_name*`,` *cells_list*`)`

This function will add a list of cells to the Cell Geometry Digest (CGD) whose access name is given as the first argument. The cells will be read using the Cell Hierarchy Digest (CHD) whose access name is given as the second argument.

This, and the `CgdRemoveCell` function can be used to implement a cache for cell data. When a CHD is used for access, and a CGD has been linked to the CHD, the CHD will read geometry information for cells in the CGD from the CGD, and cells not found in the CGD will be read from the layout file. Thus, if memory is tight, one can put only the heavily-used cells into the CGD, instead of all cells.

If the CGD already contains data for a cell to add, the data will be overwritten with the new cell data.

For the *cells_list* argument, one can pass either a handle to a list of strings that contain cell names, or a string containing space-separated cell names. If a cell named in the list is not found in the CHD, it will be silently ignored.

This applies to memory and file type CGDs, as described for `OpenCellGeomDigest`. The geometry records are saved in memory, whether or not the CGD is file type. Individual records set the access method, so it is possible to have mixed file access and memory access records in the same CGD.

On success, 1 is returned. If an error occurs, 0 is returned, and a message may be available from `GetError`.

(stringlist_handle) `CgdContents(`*cgd_name*`,` *cellname*`,` *layername*`)`

This function returns content listings from the Cell Geometry Digest (CGD) whose access name is given in the first argument. The remaining string arguments give the cell name and layer name to query. Either or both of these arguments can be null (passed 0).

If the *cellname* is null, a handle to a list if strings giving the cell names in the CGD is returned. otherwise, the *cellname* must be a cell name from the CGD.

If *layername* is null, the return value is handle to a list of layer name strings for layers used in *cellname*. If *layername* is not null, it should be one of the layer names contained in the *cellname*.

The return value when both *cellname* and *layername* are non-null is a handle to a list of two strings. The first string gives the integer number of bytes of compressed geometry for the cell/layer. The second string gives the size of the geometry string after decompression. The compressed size can be 0, in which case compression was not used as the block is too small for compression to be effective.

If the arguments are unresolved, the return value is a scalar 0.

(gs_handle) `CgdOpenGeomStream(`*cgd_name*`,` *cellname*`,` *layername*`)`

This function creates a handle to an iterator for decompressing the geometry in a Cell Geometry Digest (CGD). The first argument is the access name of the CGD. The second argument is the name of one of the cells contained in the CGD. The third argument is the name of a layer used by the cell. The cells and layers in the CGD can be listed with `CgdContents`.

The return value is a handle to an incremental reader, loaded with the compressed geometry for the cell and layer. This can be passed to `GsReadObject` to obtain the geometrical objects.

The `Close` function can be used to destroy the reader. It will be closed automatically if `GsReadObject` iterates through all objects contained in the stream.

A scalar 0 is returned if the arguments are not resolved.

(object_handle) `GsReadObject(`*gs_handle*`)`

This function takes the handle created with `CgdOpenGeomStream` and returns an object handle which points to a single object. A different object will be returned with each call until all objects have been returned, at which time the geometry stream handle is closed. Further calls will return a scalar 0.

The `ConvertReply` function can also return a handle for use by this function.

(int) `GsDumpOasisText(`*gs_handle*`)`

This function will dump the geometry stream in OASIS ASCII text representation to the console window (standard output). The handle is freed. This may be useful for debugging.

F.4.12 Assembly Stream

These functions implement a functionality similar to the **!assemble** command.

(stream_handle) `StreamOpen(`*outfile*`)`

Open an assembly stream to the file *outfile*. The file format that will be used is obtained from the extension of the name given, which must be one of

CGX	`.cgx`
CIF	`.cif`
GDSII	`.gds, .str, .strm, .stream`
OASIS	`.oas`

If successful, a handle to the stream control structure is returned, which can be passed to other functions which require this data type. A scalar zero is returned on error. The returned handle is used to implement processing of archive data similar to the **!assemble** command.

(int) `StreamTopCell(`*stream_handle*`, `*cellname*`)`
Define the name of a top-level cell that will be created in the output stream. At most one definition is possible in a stream. If successful 1 is returned, otherwise 0 is returned.

(int) `StreamSource(`*stream_handle*`, `*file_or_chd*`, `*scale*`, `*layer_filter*`, `*name_change*`)`
This function will add a source specification to a stream. The specification can refer to either an archive file, or to a Cell Hierarchy Digest (CHD). Upon successful return, the source will be queued for writing to the stream (initiated with `StreamRun`). Arguments set various modes and conditions that will apply during the write.

This function specifies the equivalent of a Source Block as described for the **!assemble** command. The `StreamInstance` function is used to add "Placement Blocks".

stream_handle
Handle to the stream object.

file_or_chd
This argument can be either a string giving a path to an archive file, or the access name of a Cell Hierarchy Digest in memory.

scale
This is a scaling factor which applies only when streaming the entire file, which will occur if no instances are specified for the source with the `StreamInstance` function. It is ignored if an instance is specified. When used, all coordinates read from the source file will be multiplied by the factor, which can be in the range 0.001 – 1000.0.

layer_filter
This is a switch integer that enables or disables use of the layer filtering and aliasing capability. If 0, no layer filtering or aliasing will be done. If nonzero, layer filtering and aliasing will be be performed when reading from the source, according to the present values of the variables listed below. These values are saved, so that the variables can subsequently change.

```
LayerList
UseLayerList
LayerAlias
UseLayerAlias
```

If needed, these variables should be set to the desired values before calling this function, then reset to the previous values after the call. This can be done with the `Get` and `Set` functions.

name_change
This is a switch integer that enables or disables use of the Cell Name Mapping capability. If 0, no cell name changes are done, except that if a name clash is detected, a new name will be supplied, similar to the auto-aliasing feature. If nonzero, cell name mapping will be performed when the source is read according to the present values of the variables listed below. These values are saved, so that the variables can subsequently change.

```
InCellNamePrefix
InCellNameSuffix
InToLower
InToUpper
```

If needed, these variables should be set to the desired values before calling this function, then reset to the previous values after the call. This can be done with the `Get` and `Set` functions.

The function returns one on success, zero otherwise with an error message probably available through `GetError`.

(int) `StreamInstance(`*stream_handle*`,` *cellname*`,` *x*`,` *y*`,` *my*`,` *rot*`,` *magn*`,` *scale*`,` *no_hier*`,` *ecf_level*`,` *flatten*`,` *array*`,` *clip*`)`
This function will add a placement name to the most recently added source file (using `StreamSource`). A source must have been specified before this function can be called successfully. This function specifies the equivalent of a Placement Block as described for the **!assemble** command.

The *cellname* must match the name of a cell found in the source, including any aliasing in effect. There are two consequences of calling this function: the named cell and possibly its subcell hierarchy will be written to output, and if a top cell was specified (with `StreamTopCell`), an instance of the named cell will be placed in the top cell. The placement is governed by the *x*, *y*, *my*, *ang*, and *magn* arguments, which are ignored if there is no top cell.

The *x*,*y* are the translation coordinates of the cell origin. The *my* is a flag indicating Y-reflection before rotation. The *ang* is the rotation angle, in degrees, and must be a multiple of 45 degrees. The *magn* is the magnification factor for the placement. These apply to the instantiation only, and have no effect on the cell definitions.

The remaining arguments affect the cell definitions that are created in the output file.

scale
This is a scale factor by which all coordinates are scaled in cell definition output, and is a real number in the range 0.001 – 1000.0. This is different from the *magn* factor, which applies only to the instance placement.

no_hier
This is a boolean value that when nonzero indicates that only the named cell, and not its hierarchy, is written to output. This can cause the output file to have unresolved references.

ecf_level
This is an integer 0–3 which specifies the empty cell filtering level as described for the **Format Conversion** panel in 14.10. The values are

0	No empty cell filtering.
1	Apply pre- and post-filtering.
2	Apply pre-filtering only.
3	Apply post-filtering only.

flatten
If the boolean variable *flatten* is nonzero, the objects in the hierarchy under *cellname* will be created in *cellname*, thus only one cell, containing all geometry, will be written.

array
If the *array* argument is passed 0, no windowing will be used. Otherwise the *array* should have four components which specify a rectangle, in microns, in the coordinates of *cellname*. The values are

array`[0]`	X left
array`[1]`	Y bottom
array`[2]`	X right
array`[3]`	Y top

If an *array* is given, only the objects and subcells needed to render the window will be written.

clip
If the boolean value *clip* is nonzero and an *array* is given, objects will be clipped to the window. Otherwise no clipping is done.

The function returns one on success, zero otherwise with an error message probably available through `GetError`.

(int) `StreamRun(`*stream_handle*`)`
This function will initiate the writing from the sources previously specified with `SteamSource` into the output file. The real work is done here. The function returns one on success, zero otherwise with an error message probably available through `GetError`.

F.5 Geometry Editing Functions 1

F.5.1 General Editing

(int) `ClearCell(`*undoable*`,` *layer_list*`)`
This function will clear the content of the present mode (electrical or physical) part of the current cell. If the first argument is nonzero, the deletions will be added to the internal undo list, otherwise not. The latter is more efficient, though this makes the deletions irreversible. The second argument, if null or empty, indicates that all objects on all layers will be deleted, including subcells. Otherwise this can be set to a string containing a space-separated list of layer names, following an optional special character '`!`' or '`^`' which must be the first character in the string if used. If the special character does not appear, the deletions apply only to the layers listed. If the special character appears, the deletions apply only to the layers *not* listed. Recall that the internal name for the layer that contains subcells ls "`$$`", thus for example using "`!  $$`" would delete all geometry but retain the subcells.

The return value is the number of objects deleted.

`Commit()`
The Commit functions terminates the present operation, adding it to the undo list. It will also redisplay any changes. This function should be called after each change or after a group of related changes. It is implicitly called when a script exits.

`Undo()`
This function will undo the most recent operation.

`Redo()`
This function will redo the last undone operation.

(int) `SelectLast(`*types*`)`
This function selects objects that have been created by the script functions since the last call to `Commit` or `SelectLast` (which calls `Commit`), according to *type*. The *type* argument is a string whose characters serve to enable selection of a given type of object: '`b`' for boxes, '`p`' for polygons, '`w`' for wires, '`l`' for labels, and '`c`' for instances. If this string is empty or null, then all objects will be selected. Objects that are created using `PressButton` or otherwise using *Xic* input implicitly call `Commit`, so can't be selected in this manner.

F.5.2 Current Transform

(int) `SetTransform(`*angle_or_string*`,` *reflection*`,` *magnification*`)`
This function sets the "current transform" to the values provided. It is similar in action to the controls in the **Current Transform** panel. The first argument can be a floating point angle that will be snapped to the nearest multiple of 45 degrees in physical mode, 90 degrees in electrical

mode. If bit 1 of *reflection* is set, a reflection of the x-axis is specified. If bit 2 of *reflection* is set, a reflection of the y-axis is specified. The *magnification* sets the scaling applied to transformed objects, and is accepted only while in physical mode. It is ignored if less than or equal to zero.

The first argument can alternatively be a string, in the format as returned from `GetTransformString`. The string will be parsed, and if no error the transform will be set. The two remaining arguments are ignored, but must be given (0 can be passed for both).

The return value is 1 on success, 0 otherwise.

Examples:

Set rotation 180, mirror the X axis:
  `SetTransform(180, 1, 1)` or `SetTransform("R180MX", 0, 0)`
Set rotation 180, mirror the Y axis:
  `SetTransform(180, 2, 1)` or `SetTransform("R180MY", 0, 0)`
Set rotation 180, mirror both X,Y axes:
  `SetTransform(180, 3, 1)` or `SetTransform("R180MYMX", 0, 0)`

(int) `StoreTransform(`*register*`)`
This function will save the current transform settings into a register, which can be recalled with `RecallTransform`. The argument is a register number 0–5. These correspond to the "last" and registers 1–5 in the **Current Transform** pop-up. This function returns 1 on success, 0 if the argument is out of range.

(int) `RecallTransform(`*register*`)`
This function will restore the transform settings previously saved with `StoreTransform`. The argument is a register number 0–5. These correspond to the "last" and registers 1–5 in the **Current Transform** pop-up. This function returns 1 on success, 0 if the argument is out of range.

(string) `GetTransformString()`
Return a string describing the current transform, an empty string will indicate the identity transform. The string is a sequence of tokens and contains no white space. It is the same format used to indicate the current transform in the *Xic* status line. The tokens are:

[`R`*ang*][`MY`][`MX`][`M`*magn*]

The square brackets indicate that each token is optional and do not appear in the string. If the rotation angle is nonzero, the first token will appear, where *ang* is the angle in degrees. This is an integer multiple of 45 degrees in physical mode, 90 degrees in electrical mode, larger than zero and smaller than 360.

If reflection of Y or X is in force, one or both of the mext two tokens will appear. These are literal. If the magnification is not unity, the final token will appear, with *magn* being a real number in the range 0.001 through 1000.0.

The order of the tokens must be as shown.

The returned string, or one in the same format, can be passed to the first argument of `SetTransform`.

(int) `GetCurAngle()`
This returns the rotation angle of the current transform, in degrees. This will be 0, 45, 90, 135, 180, 225, 270, 315 in physical mode, or 0, 90, 180, 270 in electrical mode. The `SetTransform` function can be used to set the rotation angle.

(int) `GetCurMX()`
This returns 1 if the current transform mirrors the x-axis, 0 otherwise. The `SetTransform` function can be used to set the mirror transformations.

(int) `GetCurMY()`
This returns 1 if the current transform mirrors the y-axis, 0 otherwise. The `SetTransform` function can be used to set the mirror transformations.

(real) `GetCurMagn()`
This returns the magnification component of the current transform. The `SetTransform` function can be used to set the magnification.

(int) `UseTransform(`*enable*`,` *x*`,` *y*`)`
This command enables and disables use of the current transform in the `ShowGhost` function, as well as the functions that create objects: `Box`, `Polygon`, `Arc`, `Wire`, and `Label`. The functions `Move`, `Copy`, `Logo`, and `Place` naturally use the current transform and are unaffected by this function.

All arguments are numeric. If the first argument is nonzero, the current transformation will be used in subsequent calls to the functions listed above. If the first argument is zero, the current transform is ignored by these functions. The remaining arguments provide the translation applied to the object being created, before the current transform is applied.

If `UseTransform(1, ...)` has been given, `ShowGhost` will apply the current transform to the list of objects to display, using the pointer location as the translation rather than the x, y supplied to `UseTransform`, which are ignored. The other functions listed above will create the object after applying the current transform, using x, y.

In some scripts, it will be necessary to call `UseTransform(1, ...)` twice, once to enable `ShowGhost`, and again after the location for the new object is obtained. In particular, if `Point` is used to obtain the coordinate, `UseTransform` should be called before `Point` (so the ghost drawing will be accurate) and again with the coordinates returned from `Point` before the new object is created.

The `Box` function will actually create a polygon if the current transform is being used and the rotation angle is 45 degrees or one of the other non-Manhattan angles. The `Polygon` function will actually create a box if the rotated figure can be so represented. The `Polygon` function will never create boxes unless use of the current transform is enabled.

Below is an example script that will place boxes on the current layer where the user clicks. Note that the size and rotation angle of the box can be changed while in the script through the **Transform Menu**.

```
ShowPrompt("Click to place boxes")
PushGhostBox(0, 0, 1, 1)
UseTransform(1, 0, 0)
while (1)
    ShowGhost(8)
    a[2]
    if !Point(a)
        ShowPrompt("")
        Exit()
    end
    ShowGhost(0)
    UseTransform(1, a[0], a[1])
    Box(0, 0, 1, 1)
    Commit()
end
```

F.5.3 Derived Layers

These functions provide an interface to the derived layer capability (see 15.2). Derived layers are invisible internal layers that imply geometry resulting from evaluation of a layer expression, which may involve normal layers and other derived layers. Derived layers are recognized by name in layer expressions.

There are actually two implementations of derived layer functionality. The interface functions allow explicit choice of which evaluation method to use. Within *Xic*, this detail is generally invisible to the user.

In the original implementation, developed for the DRC system, the geometry of derived layers must be created or updated before the derived layer is referenced. In use, reference to a derived layer in a layer expression retrieves this geometry, very similar to what happens when a normal layer is referenced. Ordinarily, the derived layer geometry will be cleared after final use. This method may be fast when the same layer expressions must be evaluated many times, so it seems a good match for DRC, where it is used.

To use this method, the interface function `EvalDerivedLayers` is called to create the geometry for each derived layer that will be evaluated. Then, `GetDerivedLayerLexpr` is called with a boolean true second argument to get the evaluation objects as needed, which are evaluated to create new geometry. When done, `ClearDerivedLayers` is called to destroy the precomputed geometry.

In the second mode of operation, when the parse tree for the derived layer is created, references to derived layers will be recursively parsed and stitched into the tree. The final parse tree will contain normal layers only, and can therefor be evaluated in any context, without the need for precomputed geometry caches.

With this method, there is no need to call `EvalDerivedLayers` and `ClearDerivedLayers`, as there is no use of cached geometry. The evaluation object is returned from `GetDerivedLayerLexpr` with a boolean false second argument.

(int) `AddDerivedLayer(`*lname*`,` *index*`,` *lexpr*`)`

This will add a derived layer to the database, under the name given in the first argument. The second argument is an integer layer number for the layer, which is used for ordering when the derived layers are printed, for example to an updated technology file. If not positive, *Xic* will generate a number to be used for a new layer. Numbers need not be unique, sorting is alphabetic among derived layer names with the same index number. If a derived layer of the same name already exists, it will be silently overwritten.

The third argument is a string starting with an optional keyword followed by a layer expression, separated by space. The keyword is one of `join`, `split`, or `splitv`. These are the same keywords, and have the same effects, as is explained for the DerivedLayer keyword in the technology file. The expression can reference by name ordinary layers and derived layers. The expression is not parsed until evaluation time.

The function fails if either the *lname* or *lexpr* are null or empty strings.

(int) `RemDerivedLayer(`*lname*`)`

If a derived layer exists with the given name, remove the definition from the internal registry, so that the derived layer definition and any existing geometry becomes inaccessible. The derived layer definition can be restored with `AddDerivedLayer`. If the derived layer is found and removed, this function will return 1, otherwise 0 is returned.

(int) `IsDerivedLayer(`*lname*`)`

This function will return 1 if the string argument matches a derived layer name in the database, 0 otherwise. Matching is case-insensitive.

The name can be in the form "*layer*:*purpose*" as for normal *Xic* layers, however the entire token is taken verbatim. This is a subtle difference from normal layers, where for example "`m1:drawing`" and "`m1`" are equivalent (the `drawing` purpose being the default). As derived layer names, the two would differ, and the notion of a purpose does not apply to derived layers.

(int) `GetDerivedLayerIndex(`*lname*`)`
This returns a positive integer which is the layer index number of the derived layer whose name was given, or 0 if no derived layer can be found with that name (case insensitive).

(string) `GetDerivedLayerExpString(`*lname*`)`
This returns the layer expression string for the derived layer whose name is passed. If the derived layer is not found, a null string is returned.

(layer_expr) `GetDerivedLayerLexpr(`*lname*`, `*noexp*`)`
This returns a parsed layer expression object created from the layer expression of the derived layer whose name is passed. This can be passed to other functions which can use this data type. If there is a parse error, the function fails fatally. Otherwise the return is a valid parse tree object.

The boolean second argument will suppress derived layer expansion if set.

There are two ways to handle derived layers. Generally, layer expression parse trees are expanded (second argument is false), meaning that when a derived layer is encountered, the parser recursively descends into the layer's expression. The resulting tree references only normal layers, and evaluation is straightforward.

A second approach might be faster. The parse trees are not expanded (second argument is true), and a parse node to a derived layer contains a layer descriptor, just as for normal layers. Before any computation, `EvalDerivedLayers` must be called, which actually creates database objects in a database for the derived layer. Evaluation involves only finding the geometry in the search area, as for a normal layer.

(string) `EvalDerivedLayers(`*list*`, `*array*`)`
Derived layer evaluation objects (such as the return from `GetDerivedLayerLexpr`) that are not recursively expanded must have derived layer geometry precomputed before use. This function creates derived layer geometry for this purpose.

Evaluation creates the geometry described by the layer expression. Derived layers are never visible, so this geometry is internal, but can be accessed, e.g., by design rule evaluation functions, or used to create normal layers with the **!layer** command or the **Evaluate Layer Expression** panel from the **Edit Menu**.

The first argument is a string containing a list of derived layer names, separated by commas or white space. The function will evaluate these derived layers, and any derived layers referenced in their layer expressions, in an order such that the derived layers will be evaluated before being referenced during another evaluation.

All geometry created will exist in the current cell, and the layer expressions will source all levels of the hierarchy. Any geometry left in the current cell from a previous evaluation will be cleared first. Derived layer geometry in subcells is ignored.

The second argument can set the area where the layers will be evaluated, which can be any rectangular region of the current cell. This can be an array of size four or larger, specifying left, bottom, right, and top coordinates in microns in the 0, 1, 2, 3 indices. The argument can also be a scalar 0 which indicates to use the entire current cell.

The return is a string listing all of the derived layers evaluated, which will include derived layers referenced by the original list but not included in the list. This should be passed to `ClearDerivedLayers` when finished using the layers.

(int) `ClearDerivedLayers(`*list*`)`
The argument is a string containing a list of derived layer names, separated by commas or white space. This may be the return from `EvalDerivedLayers`. All of the layers listed will be cleared in the current cell. If a layer name is not resolved as a derived layer, it is silently ignored. Clearing already clear layers is not an error. Derived layers should be cleared after their work is done, to recycle memory. The return value is an integer count of the number of derived layers that were cleared.

F.5.4 Object Management by Handles

The following functions provide a fairly complete interface to database objects.

Internally, most of the "`Set`..." functions in this group modify objects via application of the pseudo-properties (see 10.1.2). This allows modification of most objects and types, with the restrictions listed in the table below. Without restrictions, the functions can act on database objects or the "object copies" which are memory objects not part of any cell. The objects can be from electrical or physical cells, and the containing cell (if any) need not be the current cell. However, a restriction when working with copies is that the object type can not be changed.

boxes	no restrictions
polys	no restrictions
wires	can't accept electrical wires on the active (SCED) layer
labels	no restrictions
instances	can't accept electrical instances

As mentioned, some of the functions generate or accept lists of "object copies". These are objects that are not included in the object database for any cell. A list of copies behaves in most respects like an ordinary object list. The The `CopyObjects` function can be used to create a new database object from a copy. The handle manipulation functions such as `HandleCat` work, but lists of copies can *not* be mixed with lists of database objects, `HandleCat` will fail quietly if this is attempted. Copies can not be selected.

(object_handle) `ListElecInstances()`
This function returns a handle to a complete list of cell instances found in the electrical part of the current cell. Operation is identical in electrical and physical modes. In the schematic, cell instances represent subcircuits, devices, and pins. The "`GetInstance`" functions described below can be used to obtain information about the instances.

(object_handle) `ListPhysInstances()`
This function returns a handle to a complete list of cell instances found in the physical layout of the current cell. Operation is identical in electrical and physical modes. The "`GetInstance`" functions described below can be used to obtain information about the instances.

(object_handle) `SelectHandle()`
This function returns a handle to the list of objects currently selected. The list is copied internally, and so is unchanged if the objects are subsequently deselected.

A handle to the object list is returned. The `ObjectNext` function is used to advance the handle to point to the next object in the list. The `HandleContent` function returns the number of objects remaining in the list.

(object_handle) `SelectHandleTypes(`*types*`)`

This function returns a handle to a list of objects that are currently selected, but only the types of objects specified in the argument are included. The argument is a string which specifies the types of objects to include. If zero or an empty string is passed, all types are included, and the function is equivalent to `SelectHandle`. Otherwise the characters in the string signify which objects to include:

'`b`'	boxes
'`p`'	polygons
'`w`'	wires
'`l`'	labels
'`c`'	subcells

For example, passing "`pwb`" would include polygons, wires, and boxes only. The order of the characters is unimportant.

(object_handle) `AreaHandle(`*l*`,` *b*`,` *r*`,` *t*`,` *types*`)`

This function creates a list of objects that touch the rectangular area specified by the first four coordinates (which are the left, bottom, right, and top values of the rectangle). The fifth argument is a string which specifies the types of objects to include. If zero or an empty string is passed, all types are included, otherwise the characters in the string signify which objects to include:

'`b`'	boxes
'`p`'	polygons
'`w`'	wires
'`l`'	labels
'`c`'	subcells

For example, passing "`pwb`" would list polygons, wires, and boxes only. The order of the characters is unimportant.

A handle to the object list is returned. The `ObjectNext` function is used to advance the handle to point to the next object in the list. The `HandleContent` function returns the number of objects remaining in the list.

(object_handle) `ObjectHandleDup(`*object_handle*`,` *types*`)`

This function creates a new handle and list of objects. The new object list consists of those objects in the list referenced by the argument whose types are given in the string *types* argument. If zero or an empty string is passed, all types are included, otherwise the characters in the string signify which objects to include:

'`b`'	boxes
'`p`'	polygons
'`w`'	wires
'`l`'	labels
'`c`'	subcells

The return value is a handle, or 0 if an error occurred. Note that the new handle may be empty if there were no matching objects. The function will fail if the handle argument is not a pointer to an object list.

(int) `ObjectHandlePurge(`*object_handle*`,` *types*`)`

This function will purge from the list of objects referenced by the handle argument objects with types listed in the *types* string. If zero or an empty string is passed, all types are deleted, otherwise the characters in the string signify which objects to delete:

`b`	boxes
`p`	polygons
`w`	wires
`l`	labels
`c`	subcells

The return value is the number of objects remaining in the list. The function will fail if the handle argument does not reference a list of objects.

(int) `ObjectNext(`*object_handle*`)`

This function is called with a handle to a list of objects, and causes the handle to reference the next object in the list. If there are no more objects, the handle is closed, and this function returns zero. Otherwise, 1 is returned. This function will fail if the handle passed is not a handle to an object list.

(object_handle) `MakeObjectCopy(`*numpts*`,` *array*`)`

This function creates an object copy from the *numpts* coordinate pairs in the *array*. The function returns an object list handle referencing the "copy", which can be used in the same manner as copies of "real" objects. The coordinate list must be closed, i.e., the last coordinate pair must be the same as the first. If the coordinates represent a rectangle, a box object is created, otherwise the object is a polygon. Coordinates are in microns, relative to the origin of the current cell. The object is associated with the current layer (but of course it really does not exist on that layer).

(string) `ObjectString(`*object_handle*`)`

This function returns a CIF-like string describing the object pointed to by the given object handle. This provides all of the geometric information for the object. Strings of this format can be reconverted to object copies with the `ObjectCopyFromString` function.

On error or for an empty handle, a null string is returned. The function will fail if the argument is not a handle to an object list.

(object_handle) `ObjectCopyFromString(`*string*`, layer)`

This function will create an object copy from the CIF-like string, as generated by the `ObjectString` function. Boxes, polygons, and wires are supported, labels and subcells will not return a handle. The object will be associated with the layer named in the second argument. The layer will be created if it does not exist. Only physical layers are accepted.

On success, a handle to an object list containing the new copy is returned. On error, a scalar zero is returned. The function will fail if the string is null or a new layer cannot be created.

(object_handle) `FilterObjects(`*object_list*`,` *template_list*`,` *all*`,` *touchok*`,` *remove*`)`

This function creates a handle to a list of objects that is a subset of the objects contained in the *object_list*. The objects in the new list are those that touch or overlap objects in the *template_list*, which is also a handle to a list of objects.

If *all* is nonzero, all of the objects in the *template_list* will be used for comparison, otherwise only the head object in the template list will be used.

If *touchok* is nonzero, objects in the object list that touch but do not overlap the template object(s) will be added to the new list, otherwise not.

If *remove* is nonzero, objects that are added to the new list are removed from the *object_list*, otherwise the *object_list* is not touched. The function will fail if the handle arguments are of the wrong type. The return value is a new handle to a list of objects.

(object_handle) `FilterObjectsA(`*object_list*`,` *array*`,` *array_size*`,` *touchok*`,` *remove*`)`

This function creates a handle to a list of objects, which consist of the objects in the *object_list* that touch or overlap the polygon defined in the *array*. The *array_size* is the number of x-y coordinates represented in the array. In the array, the values are x-y coordinate pairs representing the polygon vertices, and the first pair must match the last pair (i.e., the figure must be closed). The values are specified in microns. If *touchok* is nonzero, objects that touch but do not overlap the polygon will be added to the list, otherwise not. If *remove* is nonzero, objects that are added to the new list are removed from the *object_list*, otherwise the *object_list* is not touched.

The function will fail if *array_size* is less than 4, or the size of the array is less than twice *array_size*, or if the handle argument is not a handle to a list of objects. The return value is a new handle to a list of objects.

(int) `CheckObjectsConnected(`*object_handle*`)`
This function returns 1 unless the list contains objects on the layer of the first object in the list that are mutually disjoint, meaning that there exist two objects and one can not draw a curve from the interior of one to the other without crossing empty area. If disjoint objects are found, 0 is returned.

(int) `CheckForHoles(`*object_handle*`,` *all*`)`
This function returns 1 if the object, or collection of objects, has "holes", i.e., uncovered areas completely surrounded by geometry. The first argument is a handle to a list of objects. If the second argument is nonzero, the geometry represented by all objects in the list is checked. If zero, only the first object (which might be a complex polygon containing holes) is checked. If no holes are found, 0 is returned.

When *all* is true, only objects on the same layer as the first object in the list are considered.

(object_handle) `BloatObjects(`*object_handle*`,` *all*`,` *dimen*`,` *lname*`,` *mode*`)`
This function returns a handle to a list of object copies which are bloated versions of the objects referenced by the handle argument, similar to the **!bloat** command. The passed handle and objects are not affected. Edges will be pushed outward or pulled inward by *dimen* (positive values push outward). The *dimen* is given in microns.

The *all* argument is a boolean that if nonzero indicates that all objects in the list referenced by the handle may be processed. If zero, only the first object in the list will be processed.

The *lname* argument is a layer name. If this argument is zero, or a null or empty string, all objects on the returned list are associated with the layer of the first object in the passed list, and only objects on this layer in the passed list are processed. Otherwise, the layer will be created if it does not exist, and all new objects will be associated with this layer, and all objects in the passed list will be processed.

The *mode* argument is an integer that specifies the algorithm to use for bloating. Giving zero specifies the default algorithm. See the description of the **!bloat** command (19.13.12) for documentation of the algorithms available.

The `DeleteObjects` function can be called to delete the old objects. The `CopyObjects` function can be called on the returned objects to add them to the database. This function returns a handle to the new list upon success, or 0 if there are no objects. The function will fail if the first argument is not a handle to a list of objects or copies, or the *lname* argument is non-null and not a valid layer name.

This function uses the JoinMax*XXX* variables in processing. There is no effect on objects in the list whose handle is passed as the first argument, or on the handle.

(object_handle) `EdgeObjects(`*object_handle*`,` *all*`,` *dimen*`,` *lname*`,` *mode*`)`
This function creates new polygon copies that cover the edges of the figures in the passed handle. The *dimen* is half the effective path width of the generated wire-like shapes that cover the edges.

If the boolean argument *all* is nonzero, all of the objects in the passed list may be processed, otherwise only the object at the head of the list will be processed.

The *lname* argument is a layer name. If this argument is zero, or a null or empty string, all objects on the returned list are associated with the layer of the first object in the passed list, and only objects on this layer in the passed list are processed. Otherwise, the layer will be created if it does not exist, and all new objects will be associated with this layer, and all objects in the passed list will be processed.

The *mode* is an integer which specifies the algorithm to use. The algorithms are described with the `EdgesZ` function.

The `DeleteObjects` function can be called to delete the old objects. The `CopyObjects` function can be called on the returned objects to add them to the database. This function returns a handle to the new list upon success, or 0 if there are no objects. The function will fail if the first argument is not a handle to a list of objects or copies, or the *lname* argument is non-null and not a valid layer name.

(object_handle) `ManhattanizeObjects(`*object_handle*`,` *all*`,` *dimen*`,` *lname*`,` *mode*`)`

This function will convert the objects pointed to by the handle argument into a list of copies, which is referenced by the returned handle. The supplied objects and handle are not affected. Each new object is a Manhattan approximation of the original object. The *dimen* argument is the minimum height or width in microns of rectangles created to approximate the non-Manhattan parts.

The *all* argument is a boolean that if nonzero indicates that all objects in the list referenced by the handle may be processed. If zero, only the first object in the list will be processed.

The *lname* argument is a layer name, or zero. If a layer name is given, the new objects will be associated with that layer, which will be created if it does not exist. If 0 or an empty string is passed, the new objects will be associated with the layer of the original object.

The *mode* argument is a boolean value which selects one of two Manhattanizing algorithms to employ. These algorithms are described with the **!manh** command.

The function will fail if the first argument is not a handle to a list of objects or copies, or the *lname* argument is non-null and not a valid layer name, or the *dimen* argument is smaller than 0.01. On success, a handle to the list of copies is returned. Each object in the returned list is a box or Manhattan polygon which approximates one of the original objects. Of course, if the original objects were all Manhattan, the shapes will be unchanged, though the coordinates will be moved to a *dimen* grid if the gridding mode (*mode* nonzero) is given.

The `DeleteObjects` function can be called to delete the old objects. The `CopyObjects` function can be called on the returned objects to add them to the database.

This function uses the JoinMax*XXX* variables in processing. There is no effect on objects in the list whose handle is passed as the first argument, or on the handle.

(int) `GroupObjects(`*object_handle*`,` *array*`)`

This function acts on the first object in the list and all other objects on the same layer found in the list. The objects are copied, then sorted into groups, so that each group forms a single figure, i.e., no two members of the same group are disjoint. The groups are then joined into polygons, and a handle to each group is returned in the array. The array will be resized if necessary. The returned value is the number of groups, corresponding to the used entries in the array. The `H` function should be used on the array elements to convert the values to an object handle data type, similar to the treatment of the array returned from the `HandleArray` function. The `CloseArray` function can be used to close the handles. The created objects are copies, so are not added to the database.

This function uses the JoinMax*XXX* variables in processing. There is no effect on objects in the list whose handle is passed as the first argument, or on the handle. The value 0 is returned on error or if the list is empty.

(object_handle) `JoinObjects(`*object_handle*`,` *lname*`)`
This function will combine the objects in the list passed as the first argument, if possible, into a new list of object copies, which is returned. The passed handle and objects are not affected. All objects in the returned list will be associated with the layer named in the second argument. This layer will be created if it does not exist, and the output will consist of the joined outlines of all of the objects in the passed list, from any layer. If 0, or a null or empty string is passed, the new objects will be associated with the layer of the first object in the passed list, and only the outlines of objects on this layer found in the passed list will contribute to the result.

The `DeleteObjects` function can be called to delete the old objects. The `CopyObjects` function can be called on the returned objects to add them to the database. This function returns a handle to the new list upon success, or 0 if there are no objects. The function will fail if the first argument is not a handle to a list of objects or copies, or the *lname* argument is non-null and not a vail layer name.

This function uses the JoinMax*XXX* variables in processing. There is no effect on objects in the list whose handle is passed as the first argument, or on the handle.

(object_handle) `SplitObjects(`*object_handle*`,` *all*`,` *lname*`,` *vert*`)`
This function will split the objects in the list passed as the first argument into horizontal or vertical trapezoids (polygons or boxes) and return a list of the new objects. The new objects are "object copies" and are not added to the database.

If the boolean argument *all* is nonzero, all of the objects in the list referenced by the handle will be processed. Otherwise, only the first object will be processed.

The new objects are placed on the layer with the name given in *lname*, which is created if it does not exist, independent of the originating layer of the objects. If a null string or 0 is passed for *lname*, the target layer will be the layer of the first object found in the object list.

The *vert* argument is an integer which if nonzero indicates a vertical decomposition, otherwise a horizontal decomposition is produced.

The handle and objects passed are untouched. The `DeleteObjects` function can be called to delete the old objects. The `CopyObjects` function can be called on the returned objects to add them to the database. This function returns a handle to the new list upon success, or 0 if there are no objects. The function will fail if the first argument is not a handle to a list of objects or copies, or the *lname* argument is non-null and not a valid layer name.

(int) `DeleteObjects(`*object_handle*`,` *all*`)`
Calling this function will delete referenced objects from the current cell. If the boolean argument *all* is nonzero, all objects in the list will be deleted. Otherwise, only the first object in the list will be deleted. Once deleted, the objects are no longer referenced by the handle, which may become empty as a result.

This function will fail if the handle passed is not a handle to an object list. The number of objects deleted is returned.

(int) `SelectObjects(`*object_handle*`,` *all*`)`
This function will select objects referenced by the handle. If the boolean argument *all* is nonzero, all objects in the list will be selected. Otherwise, only the first object in the list will be selected.

It is not possible to select object copies, 0 is returned if the passed handle represents copies. Otherwise the return value is the number of newly selected objects.

This function will fail if the handle passed is not a handle to an object list.

(int) `DeselectObjects(`*object_handle*`,` *all*`)`
This function will deselect objects referenced by the handle. If the boolean argument *all* is nonzero, all objects in the list will be deselected. Otherwise, only the first object in the list will be deselected.

It is not possible to select object copies, 0 is returned if the passed handle represents copies. Otherwise the return value is the number of newly deselected objects.

This function will fail if the handle passed is not a handle to an object list.

(int) `MoveObjects(`*object_handle*`,` *all*`,` *refx*`,` *refy*`,` *x*`,` *y*`)`
This function is similar to the `Move` function, however it operates on the object(s) referenced by the handle. An object is moved such that the coordinate *refx*, *refy* is translated to *x*, *y*. The current transform will be applied to the move. If *all* is nonzero, all objects in the list are moved, otherwise only the object currently referenced is moved. The function returns the number of objects moved. This function will fail if the handle passed is not a handle to an object list.

If the handle references object copies, each copy is translated and possibly transformed as described above. The handle will subsequently reference the modified object.

(int) `MoveObjectsToLayer(`*object_handle*`,` *all*`,` *refx*`,` *refy*`,` *x*`,` *y*`,` *oldlayer*`,` *newlayer*`)`
This is similar to the `MoveObjects` function, but allows layer change. If *newlayer* is 0, null, or empty, *oldlayer* is ignored and the function behaves identically to `MoveObjects`. Otherwise the *newlayer* string must be a layer name. If *oldlayer* is 0, null, or empty, all moved objects are placed on *newlayer*. Otherwise, *oldlayer* must be a layer name, in which case only objects on *oldlayer* will be placed on *newlayer*, other objects will remain on the same layer. Subcell objects are moved as in `MoveObjects`, i.e., the layer arguments are ignored.

(int) `CopyObjects(`*object_handle*`,` *all*`,` *refx*`,` *refy*`,` *x*`,` *y*`,` *repcnt*`)`
This function is similar to the `Copy` function, however it operates on the object(s) referenced by the handle. An object is copied such that the coordinate *refx*, *refy* is translated to *x*, *y*.

The *repcnt* is an integer replication count in the range 1–100000, which will be silently taken as one if out of range. If not one, multiple copies are made, at multiples of the translation factors given.

The current transform will be applied to the copy. If *all* is nonzero, all of the objects in the list are copied, otherwise only the object currently being referenced is copied. The function returns the number of objects copied. This function will fail if the handle passed is not a handle to an object list.

If the handle references object copies, the object copies that are referenced remains untouched, however the new objects, translated and possibly transformed as described above, are added to the database. The *repcnt* argument is ignored in this case.

(int) `CopyObjectsToLayer(`*object_handle*`,` *all*`,` *refx*`,` *refy*`,` *x*`,` *y*`,` *oldlayer*`,` *newlayer*`,` *repcnt*`)`
This is similar to the `CopyObjects` function, but allows layer change. If *newlayer* is 0, null, or empty, *oldlayer* is ignored and the function behaves identically to `CopyObjects`. Otherwise the *newlayer* string must be a layer name. If *oldlayer* is 0, null, or empty, all copied objects are placed on *newlayer*. Otherwise, *oldlayer* must be a layer name, in which case only objects on *oldlayer* will be placed on *newlayer*, other objects will remain on the same layer. Subcell objects are copied as in `CopyObjects`, i.e., the layer arguments are ignored.

(object_handle) `CopyObjectsH(`*object_handle*`,` *all*`,` *refx*`,` *refy*`,` *x*`,` *y*`,` *oldlayer*`,` *newlayer*`,` *todb*`)`
This function returns an object handle, containing copies of the objects in the handle passed as the first argument. If boolean *all* is set, all passed objects will be copied, otherwise only the first

object in the list will be copied. The next four arguments set the copy translation, with *refx* and *refy* in the passed object translated to *x*, *y* in the copy. The current transform is also applied to the copy.

The two layer name arguments behave as in `CopyObjectToLayer`. If *newlayer* is 0, null, or empty, *oldlayer* is ignored and no object layers will change. Otherwise the *newlayer* string must be a layer name. If *oldlayer* is 0, null, or empty, all copied objects are placed on *newlayer*. Otherwise, *oldlayer* must be a layer name, in which case only objects on *oldlayer* will be placed on *newlayer*, other objects will remain on the same layer. Subcell objects are copied as in `CopyObjects`, i.e., the layer arguments are ignored.

The final argument is a boolean that when true, the copies are added to the database, and the returned handle points to the database objects. If false, the returned handle contains "object copies" which do not appear in the database. Note that when copies are added to the database, unlike other copy functions merging is disabled, and the replication feature is not available.

(string) `GetObjectType(`*object_handle*`)`

This function returns a one-character string representing the type of object referenced by the handle argument. If the handle is invalid, a null string is returned. The types are:

'`b`'	boxes
'`p`'	polygons
'`w`'	wires
'`l`'	labels
'`c`'	subcells

This function will fail if the handle passed is not a handle to an object list.

(int) `GetObjectID(`*object_handle*`)`

This function returns a unique id number for the object. The id is actually the address of the object in the process memory, so it is valid only for the current *Xic* process. If the referenced object is a copy, the id returned is the address of the real object, not the copy. If no object is referenced by the handle, 0 is returned. The function fails if the handle is not an object list type.

(int) `GetObjectArea(`*object_handle*`)`

Return the area in square microns of the object pointed to by the handle. Zero is returned for a defunct handle or upon error.

(int) `GetObjectPerim(`*object_handle*`)`

Return the perimeter in microns of the object pointed to by the handle. Zero is returned for a defunct handle or upon error.

(int) `GetObjectCentroid(`*object_handle*`, `*array*`)`

Return the centroid coordinates in microns of the object pointed to by the handle. The second argument is an array of size two or larger that will contain the centroid coordinates upon successful return. The return value is zero for a defunct handle or upon error, one if success.

(int) `GetObjectBB(`*object_handle*`, `*array*`)`

This function loads the left, bottom, right, and top coordinates of the object's bounding box (in microns) into the *array* passed. This function will fail if the handle passed is not a handle to an object list, or if the size of the array is less than 4. The return value is 1 if successful, 0 otherwise.

(int) `SetObjectBB(`*object_handle*`, `*array*`)`

This function will alter the shape of the object pointed to by the handle such that it has the bounding box passed. The *array* contains the left, bottom, right, and top coordinates, in microns.

This function will fail if the handle passed is not a handle to an object list, or if the size of the array is less than 4. The return value is 1 if successful, 0 otherwise. This function has no effect on subcells, but other types of object will be rescaled to the new bounding box.

(int) `GetObjectListBB(`*object_handle*`,` *array*`)`

This is similar to `GetObjectBB`, but computes the bounding box of all objects in the list of objects referenced by the handle. not just the list head. The function loads the left, bottom, right, and top coordinates of the aggregate bounding box (in microns) into the array passed. This function will fail if the handle passed is not a handle to an object list, or if the size of the array is less than 4. The return value is a count of the objects in the list.

(int) `GetObjectXY(`*object_handle*`,` *array*`)`

This function will retrieve the "XY" position from the object pointed to by the handle into the array, which must have size 2 or larger. This is a coordinate, in microns, the interpretation of which depends on the object type. For boxes, that value is the lower-left corner of the box. For wires and polygons, the value is the first vertex in the coordinate list. For labels, the value is the text anchor position. For subcells, the value is the instantiation point, the same as the translation in the instantiation transform.

On success, the return value is 1, with the array values set. Otherwise, 0 is returned.

(int) `SetObjectXY(`*object_handle*`,` *x*`,` *y*`)`

This function will set the "XY" coordinate of the object pointed to by the handle, as if setting the XprpXY pseudo-property number 7215 on the object. This has the effect of moving the object to a new location. The interpretation of the coordinate, which is supplied in microns, depends on the type of object. For boxes, the lower-left corner will assume the new value. For polygons and wires, the object will be moved so that the first vertex in the coordinate list will assume the new value. For labels, the text will be anchored at the new value, and for subcells, the new value will set the translation part of the instantiation transform.

A value of 1 is returned if the operation succeeds, and the object will be moved. On failure, 0 is returned.

(string) `GetObjectLayer(`*object_handle*`)`

This function returns the name of the layer on which the object referenced by the handle is defined. For subcells, this layer is named "`$$`", but objects will return a layer from the layer table. This function will fail if the handle passed is not a handle to an object list. A stale handle will return a null string.

(int) `SetObjectLayer(`*object_handle*`,` *layername*`)`

This function will move the object to the layer named in the string *layername*. This will have no effect on subcells. A value 1 is returned if successful, 0 otherwise. This function will fail if the handle passed is not a handle to an object list.

(int) `GetObjectFlags(`*object_handle*`)`

This function returns internal flag data from the object referenced by the handle. This function will fail if the handle passed is not a handle to an object list. A stale handle will return 0.

The following flags are defined:

Name	Bit	Description
`MergeDeleted`	`0x1`	Object has been deleted due to merge.
`MergeCreated`	`0x2`	Object has been created due to merge.
`NoDRC`	`0x4`	Skip DRC tests on this object.
`Expand`	`0x8`	Five flags are used to keep track of cell expansion in main plus four sub-windows, in cell instances only.
`Mark1`	`0x100`	General purpose application flag.
`Mark2`	`0x200`	General purpose application flag.
`MarkExtG`	`0x400`	Extraction system, in grouping phonycell.
`MarkExtE`	`0x800`	Extraction system, in extraction phonycell.
`InQueue`	`0x1000`	Object is in selection queue.
`NoMerge`	`0x4000`	Object will not be merged.
`IsCopy`	`0x8000`	Object is a copy, not in database.

The bitwise logic functions such as `AndBits` can be used to check the state of the flags. Of these, only `NoDRC`, `Mark1`, and `Mark2` can be arbitrarily set by the user, using functions described below.

(int) `SetObjectNoDrcFlag(`*object_handle*`,` *value*`)`
This will set the state of the `NoDRC` flag of the object referenced by the handle. The second argument is a boolean representing the flag state. This can be called on any object, but is only significant for boxes, polygons, and wires in the database. Objects with this flag set are ignored during design rule checking.

The return value is 0 or 1 representing the previous state of the flag, or -1 on error.

(int) `SetObjectMark1Flag(`*object_handle*`,` *value*`)`
This will set the state of the `Mark1` flag of the object referenced by the handle. The second argument is a boolean representing the flag state. This can be called on any object. The flag is unused by *Xic*, but can be set and tested by the user for any purpose. The flag persists as long as the object is in memory.

The return value is 0 or 1 representing the previous state of the flag, or -1 on error.

(int) `SetObjectMark2Flag(`*object_handle*`,` *value*`)`

This will set the state of the `Mark2` flag of the object referenced by the handle. The second argument is a boolean representing the flag state. This can be called on any object. The flag is unused by *Xic*, but can be set and tested by the user for any purpose. The flag persists as long as the object is in memory.

The return value is 0 or 1 representing the previous state of the flag, or -1 on error.

(int) `GetObjectState(`*object_handle*`)`
This function returns a status value for the object referenced by the handle. The status values are:

0	normal state
1	object is selected
2	object is deleted
3	object is incomplete
4	object is internal only

Only values 0 and 1 are likely to be seen. This function will fail if the handle passed is not a handle to an object list. A stale handle will return 0.

(int) `GetObjectGroup(`*object_handle*`)`

This function returns the conductor group number of the object, which is a non-negative integer or possibly -1 in certain cases, and is assigned internally by the extraction system. This is used by the extraction system to establish connectivity nets of boxes, polygons, and wires, and for subcell indexing. If extraction is unavailable or not being used, then an arbitrary integer can be applied for other uses with the `SetObjectGroup` function.

This function will fail if the handle passed is not a handle to an object list. If no group has been assigned, or the handle is stale, or the object is part of the "ground" group, 0 is returned. Otherwise, any assigned number will be returned.

(int) `SetObjectGroup(`*object_handle*`, `*group_num*`)`

This function will assign the group number to the object. All objects and instances may receive a group number, which is an arbitrary integer. The group number is usually assigned and used by the extraction system, and should **not** be assigned with this function if extraction is being used. However, if extraction is unavailable or not being used, then this function allows an arbitrary integer to be associated with an object, which might be useful. Beware that this number is zeroed if the object is modified, or in copies.

The `GetObjectGroup` function can be used to obtain the group number of an object or cell instance.

This function will fail if the handle passed is not a handle to an object list. If the group number is successfully assigned, 1 is returned, 0 is returned otherwise.

(int) `GetObjectCoords(`*object_handle*`, `*array*`)`

This function will obtain the vertex list for polygons and wires, or the bounding box vertices of other objects, starting from the lower left corner and working clockwise. If an array is passed, the vertex coordinates are copied into the array, and the vertex count is returned. The array will contain the x, y values of the vertices, in microns, if successful. The coordinates are copied only if the array is large enough, or can be resized. If the array is a pointer to a too small array, or the array is too small but has other variables pointing to it, resizing is impossible and the copying is skipped. In this case, the returned value is the negative vertex count. If 0 is passed instead of the array, the (positive) vertex count is returned. Zero is returned if there is an error. This function will fail if the handle passed is not a handle to an object list.

(int) `SetObjectCoords(`*object_handle*`, `*array*`, `*size*`)`

This function will modify a physical object to have the vertex list passed in the array. The size is the number of vertices (one half the size of the array used). For all but wires, the first and last vertices must coincide, thus the minimum number of vertices is four. The array consists of x, y coordinates of the vertices. If the operation is successful, 1 is returned, otherwise 0 is returned. The coordinates in the array are in microns. If the coordinates represent a rectangle, the new object will be a box, if it was previously a polygon or box. A box may be converted to a polygon if the coordinates are not those of a rectangle. For labels, the coordinates must represent a rectangle, and the label will be stretched to the new box. The function has no effect on instances. This function will fail if the handle passed is not a handle to an object list.

(real) `GetObjectMagn(`*object_handle*`)`

This function returns the magnification part of the transform if the object referenced by the handle is a subcell, or 1.0 for other objects. Only physical subcells can have non-unit magnification. This function will fail if the handle passed is not a handle to an object list. A stale handle returns 0.

(int) `SetObjectMagn(`*object_handle*`, `*magn*`)`

This will set the magnification of the subcell referenced by the handle, or scale other physical objects. The real number *magn* must be between .001 and 1000 inclusive. If the operation is successful, 1 is returned, otherwise 0 is returned. This function will fail if the handle passed is not a handle to an object list.

(real) `GetWireWidth(`*object_handle*`)`
This function will return the wire width if the object referenced by the handle is a wire, otherwise 0 is returned. This function will fail if the handle passed is not a handle to an object list.

(int) `SetWireWidth(`*object_handle*`,` *width*`)`
This function will set the width of the wire referenced by the handle to the given *width* (in microns). If the operation is successful, 1 is returned, otherwise 0 is returned. This function will fail if the handle passed is not a handle to an object list.

(int) `GetWireStyle(`*object_handle*`)`
This function returns the end style code of the wire pointed to by the handle, or -1 if the object is not a wire. The codes are

0	flush ends
1	projecting rounded ends
2	projecting square ends

This function will fail if the handle passed is not a handle to an object list.

(int) `SetWireStyle(`*object_handle*`,` *code*`)`
This function will change the end style of the wire referenced by the handle to the given *code*. The code is an integer which can take the following values

0	flush ends
1	projecting rounded ends
2	projecting square ends

If the operation succeeds, 1 is returned, otherwise 0. This can apply to physical wires only. This function will fail if the handle passed is not a handle to an object list.

(int) `SetWireToPoly(`*object_handle*`)`
This function converts the wire object referenced by the handle to a polygon object. If the conversion is done, the handle will reference the new polygon object. The conversion will be done only if the wire has nonzero width. If the wire is not a copy, the wire object in the database will be converted to a polygon. Otherwise, only the copy will be changed. Upon success, the function returns 1, otherwise 0 is returned. The function fails if the argument is not a handle to an object list.

(int) `GetWirePoly(`*object_handle*`,` *array*`)`
This function returns the polygon used for rendering a wire. This will be different from the wire vertices, if the wire has nonzero width. The first argument is a handle to an object list which references a wire object. The second argument is an array which will hold the polygon coordinates. This argument can be 0, if the polygon points are not needed. The array will be resized if necessary (and possible). The return value is the number of vertices required or used in the polygon. If an error occurs, the return value is 0. If an array is passed which can't be resized because it is referenced by a pointer, the return value is a negative value, the negative vertex count required. The function will fail if the first argument is not a handle to an object list, or the second argument is not an array or zero. The coordinates returned in the array are in microns, relative to the origin of the current cell.

(string) `GetLabelText(`*object_handle*`)`
This function returns the label text if the object referenced by the handle is a label. Otherwise,

a null string is returned. The actual text is always returned, and not the symbolic text that is shown on-screen for script and long text labels. This function will fail if the handle passed is not a handle to an object list.

(int) `SetLabelText(`*object_handle*`,` *text*`)`
This function will set the label text of a label referenced by the handle. Setting the text in this manner will cause a long-text label to revert to a normal label. If the operation succeeds, the return value is 1, otherwise 0 is returned. This function will fail if the handle passed is not a handle to an object list.

(int) `GetLabelFlags(`*object_handle*`)`
This function returns the flags word used to specify a number of label presentation attributes, as described in C.2.

This function will fail if the handle passed is not a handle to an object list.

The function was named `GetLabelXform` in releases prior to 3.3.1, and is still recognized by that name, though this is deprecated and undocumented.

(int) `SetLabelFlags(`*object_handle*`,` *flags*`)`
This function will apply the given flags to the label referenced by the handle. The flags are the label flags used by *Xic* and described in C.2. If the operation is successful, 1 is returned, otherwise 0 is returned. This function will fail if the handle passed is not a handle to an object list.

The function was named `SetLabelXform` in releases prior to 3.3.1, and is still recognized by that name, though this is deprecated and undocumented.

(int) `GetInstanceArray(`*object_handle*`,` *array*`)`
This function fills in the *array*, which must have size of four or larger, with the array parameters for the instance referenced by the handle. If the operation succeeds, 1 is returned, and the array components have the following values, relative to the untransformed coordinates:

array[0]	number of cells along x
array[1]	number of cells along y
array[2]	center to center x spacing (in microns)
array[3]	center to center y spacing (in microns)

If the operation fails, 0 is returned. This function will fail if the handle passed is not a handle to an object list.

(int) `SetInstanceArray(`*object_handle*`,` *array*`)`
This function will change the array parameters of the instance referenced by the handle to the indicated values. The *array* values are in the format as returned from `GetInstanceArray`. Only physical mode subcells can be changed by this function, arrays are not supported in electrical mode. If the operation succeeds, 1 is returned, otherwise 0 is returned. This function will fail if the handle passed is not a handle to an object list.

(string) `GetInstanceXform(`*object_handle*`)`
This function returns a string giving the CIF transformation code for the instance referenced by the handle. If the object is not an instance, a null string is returned. This function will fail if the handle passed is not a handle to an object list.

(string) `GetInstanceXformA(`*object_handle*`,` *array*`)`
This function fills in the *array*, which must have size 4 or larger, with the components of the transformation of the instance referenced by the handle. The values are:

array`[0]`	1 if mirror-y, 0 if no mirror-y
array`[1]`	angle in degrees
array`[2]`	translation x
array`[3]`	translation y

This is the same data as provided by the `GetInstanceXform` function, but in numerical rather than string form. The transform components are applied in the order as found in the array, i.e., mirror first, then rotate, then translate. The function returns 1 if successful, 0 otherwise. It will fail if the handle passed is not a handle to an object list.

(int) `SetInstanceXform(`*object_handle*`,` *transform*`)`

This function applies the given *transform* to the instance referenced by the handle. The *transform* is in the form of a CIF transformation string, as returned by `GetInstanceXform`. Note that coordinates in the transform string are in internal units (1 unit = .001 micron). Only physical-mode subcells can be modified by this function. If the operation succeeds, 1 is returned, otherwise 0 is returned. This function will fail if the handle passed is not a handle to an object list.

(int) `SetInstanceXformA(`*object_handle*`,` *array*`)`

This function applies the given transform parameters in the *array* to the instance referenced by the handle. The parameters are:

array`[0]`	1 if mirror-y, 0 if no mirror-y
array`[1]`	angle in degrees
array`[2]`	translation x
array`[3]`	translation y

Only physical-mode subcells can be modified by this function. If the operation succeeds, 1 is returned, otherwise 0 is returned. The transform components are applied in the order as found in the array, i.e., mirror first, then rotate, then translate. The function returns 1 if successful, 0 otherwise. It will fail if the handle passed is not a handle to an object list.

(string) `GetInstanceMaster(`*object_handle*`)`

Note: prior to 4.2.12, this function was called `GetInstanceName`.

This function returns the master cell name of the instance referenced by the handle. If the object is not an instance, a null string is returned. This function will fail if the handle passed is not a handle to an object list. The cell instance can be electrical or physical, and operation is identical in electrical and physical mode.

(int) `SetInstanceMaster(`*object_handle*`,` *newname*`)`

Note: prior to 4.2.12, this function was called `SetInstanceName`.

This currently works with physical cell data only.

This function will replace the instance referenced by the handle with an instance of the cell given as *newname*, in the parent cell of the referenced instance. The current transform is added to the transform of the new instance. This function will fail if the handle passed is not a handle to an object list. If successful, 1 is returned, otherwise 0 is returned.

(string) `GetInstanceName(`*object_handle*`)`

Note: prior to 4.2.12, this function returned the name of the instance master cell. The `GetInstanceMaster` function now performs that operation.

This function returns a name for the electrical cell instance referenced by the handle. This is the name of the object, as would appear in a generated SPICE file.

For unnamed (missing name property) electrical instances, a null string is returned.

For physical cell instances, an instance name is returned, which consists of the master name followed by a colon separator and an index number. The index is a 0-based sequence for instances with a particular master. The index count advances by the size of the array for arrayed instances, leaving room in the sequence for individual elements. The index is in database order (top to bottom then left to right of the upper left corner of the instance bounding box), and is stable and reproducible as long as instance sizes and placement locations remain the same.

Internally, electrical names are generated in the following way. Each device has a prefix, as specified in the technology file. The prefix for subcircuits is "X", which is defined internally. The prefixes follow (or should follow) SPICE conventions. The database of instance placements is scanned in order of the placement location (upper-left corner of the instance bounding box) top to bottom, then left to right. Each instance encountered is given an index number as a count of the same prefix previously encountered in the scan. The prefix followed by the index forms the instance name. This will identify each instance uniquely, and the sequencing is predictable from spatial location in the schematic. For example. `X1` will be above or to the left of `X2`.

Rather than the internal electrical name. this function will return an assigned name, if one has been given using `SetInstanceName` or by setting the name property,

The index number can be obtained as an integer with `GetInstanceIdNum`. See also `GetInstanceAltName` for a different subcircuit name style.

(int) `SetInstanceName(`*object_handle*`,` *newname*`)`
Note: prior to 4.2.12, this function would re-master the instance, the same as the present `SetInstanceMaster` function.

This will set a name for the electrical instance referenced by the handle, which is in effect applying a name property to the instance. this makes sense for devices, subcircuits, and terminal devices. The new name will be used when generating netlist output, so should conform to any requirements, for example SPICE conventions, being in force.

If the string is null or 0, any applied name will be deleted, equivalent to "removing" a name property.

Physical instance names can not be changed, an attempt to do so fails silently.

The return value is 1 on success, 0 otherwise.

(string) `GetInstanceAltName(`*object_handle*`)`
This returns an alternative instance name for the electrical subcircuit cell instance referenced by the handle. The format is the master cell name, followed by an underscore, followed by an integer. The integer is zero-based and sequential among instances of a given master. For example, instances of master "`foo`" would have names `foo_0`, `foo_1`, etc. This is more useful an some cases than the SPICE-style names `X1`, `X2`, ... as returned by `GetInstanceName`.

For electrical device instances, this function returns the same name As the `GetInstanceName` function.

The `GetInstanceAltIdNum` function returns the index number used, as an integer. This is different from the regular index, where every instance, of whatever type, has a unique index. Here, instances of each master each have an index count starting from zero. The order that instances appear, however, is the same in both lists.

Presently, this function returns a null string for physical instances.

(string) `GetInstanceType(`*object_handle*`)`
This function will return a string consisting of a single letter that indicates the type of cell instance referenced by the handle. The function will fail if the handle is of the wrong type. A null string is returned it the object referenced is not a cell instance. Otherwise, the following strings may be returned.

These apply to electrical cell instances.

"b"
: The instance is "bad". There has been an error.

"n"
: The instance type is "null" meaning that it has no electrical significance in a schematic.

"g"
: The instance is a ground pin. It has a "hot spot" that when placed forces a ground contact at that location.

"t"
: This is a terminal device, which has a name label and hot spot. When placed, it forces a contact to a net named in the label at the hot spot location.

"d"
: The instance represents a device, such as a resistor, capacitor, or transistor.

"m"
: This is a macro, which implements a subcircuit that is placed in the schematic, as a "black box". Unlike a subcircuit, a macro has no sub-structure.

"s"
: This is an instance of a circuit cell, i.e., a subcircuit. Its master contains instances of devices and other objects representing a circuit.

For physical instances, at present there is only one return.

"p"
: This is a physical instance.

(int) `GetInstanceIdNum(`*object_handle*`)`

This function returns the integer index number used in electrical device and subcircuit instance names. See the `GetInstanceName` description for information about how the numbers are computed. Each subcircuit will have a unique number. Devices are numbered according to their prefix strings, each unique prefix has its own number sequence. These values are always non-negative.

The return for all physical instances is similarly created, and is the same index used in the instance name returned by `GetInstanceName`.

This function will return -1 on error.

(int) `GetInstanceAltIdNum(`*object_handle*`)`

This returns an alternative index for electrical subcircuits, as used in the `GetInstanceAltName` function. Every subcircuit master will have its instances numbered sequentially starting with 0. The ordering is set by the instance placement location in the schematic, top to bottom then left to right, with the upper-left corner of the bounding box being the reference location.

For physical instances, an internal indexing number used by the extraction system is returned. This is a unique 0-based sequence applied to all instances of a cell, in database order. The count is incremented by the array size for arrayed instances.

For other instances, the return value is the same as `GetInstanceIdNum`.

F.6 Geometry Editing Functions 2

F.6.1 Cells, PCells, Vias, and Instance Placement

(int) `CheckPCellParam(`*library*`, `*cell*`, `*view*`, `*pname*`, `*value*`)`

The first three arguments specify a parameterized cell. If *library* is not given as a scalar 0, it is the name of the OpenAccess library containing the pcell super-master, whose name is given in the *cell* argument. The *view* argument can be passed a scalar 0 to indicate that the OpenAccess view name is "`layout`", or the actual view name can be passed if different. For *Xic* native pcells not stored in OpenAccess, the *library* and *view* should both be 0 (zero).

The *pname* is a string containing a parameter name for a parameter of the specified pcell, and the *value* argument is either a scalar or string value. The function returns 1 if the value is not forbidden by a constraint, 0 otherwise.

(int) `CheckPCellParams(`*library*`,` *cell*`,` *view*`,` *params*`)`
The first three arguments specify a parameterized cell. If *library* is not given as a scalar 0, it is the name of the OpenAccess library containing the pcell super-master, whose name is given in the *cell* argument. The *view* argument can be passed a scalar 0 to indicate that the OpenAccess view name is "`layout`", or the actual view name can be passed if different. For *Xic* native pcells not stored in OpenAccess, the *library* and *view* should both be 0 (zero).

The *params* argument is a string providing the parameter values in the format of the pc_params property as applied to sub-masters and instances. i.e., values are constants and constraints are not included. The function returns 1 if no parameter has a value forbidden by a constraint, 0 otherwise.

(int) `CreateCell(`*cellname*`, [`*orig_x*`,` *orig_y*`])`
This will create a new cell from the contents of the selection queue, with the given name, which can not already be in use. The new cell is created in memory only, with the modified flag set so as to generate a reminder to the user to save the cell to disk when exiting *Xic*. This provides functionality similar to the **Create Cell** button in the **Edit Menu**.

If the optional coordinate pair *orig_x* and *orig_y* are given (in microns), then this point will be the new cell origin in physical mode only. Otherwise, the lower-left corner of the bounding box of the objects will be the new cell origin. In electrical mode, the cell origin is selected to keep contacts on-grid, and the origin arguments are ignored.

By default, this function will fail if a cell of the same name already exists in the current symbol table. However, if the CrCellOverwrite variable is set, existing cells will be overwritten with the new data, and the function will succeed.

(int) `CopyCell(`*name*`,` *newname*`)`
This function will copy the cell in memory named *name* to *newname*. The function returns 1 if the operation was successful, 0 otherwise. The *name* cell must exist in memory, and the *newname* can not clash with an existing cell or library device.

(int) `RenameCell(`*oldname*`,` *newname*`)`
This function will rename the cell in memory named *oldname* to *newname*, and update all references. The function returns 1 if the operation was successful, 0 otherwise. The *oldname* cell must exist in memory, and the *newname* can not clash with an existing cell or library device.

(int) `DeleteEmpties(`*recurse*`)`
This function will delete empty cells found in the hierarchy under the current cell. This operation can not be undone. The argument is an integer flag; if zero, one pass is done, and all empty cells are deleted. If the argument is nonzero, additional passes are done to delete cells that are newly empty due to their subcells being deleted on the previous pass. The top-level cells is never deleted. The return value is the number of cells deleted.

(int) `Place(`*cellname*`,` *x*`,` *y* `[,` *refpt*`,` *array*`,` *smash*`,` *usegui*`,` *tfstring*`])`
This function places an instance of the named cell at x, y. The first argument is of string type and contains the name of the cell to place. The string can consist of two space-separated words. If so,

the first word may be a CHD name, an archive file name, or a library name (including OpenAccess when available).

The interpretation is similar to the **new** selection in the **Open** command in the **File Menu**. In the case of two words, the second word is the name of the cell to extract from the source specified as the first word. If only one word is given, it can be an archive file name in which case the top-level cell is understood, or a CHD name in which case the default cell is understood, or it can be the name of a cell available as a native cell from a library or the search path, or already exist in memory.

The second two arguments define the placement location, in microns.

The remaining arguments are optional, meaning that they need not be given, but all arguments to the left must be given.

The *refpt* argument is an integer code that specifies the reference point which will correspond to x, y after placement. The values can be

0	the cell origin (the default)
1	the lower left corner
2	the upper left corner
3	the upper right corner
4	the lower right corner

The corners are those of the untransformed array or cell.

In electrical mode, if the cell has terminals, this code is ignored, and the location of the first terminal is the reference point. If the cell has no terminals, the corner reference points are snapped to the nearest grid location. This is to avoid producing off-grid terminal locations.

The *array* argument, if given, can be a scalar, or the name of an array containing four numbers. This argument specifies the arraying parameters for the instance placement, which apply in physical mode only. If a scalar 0 is passed, the placement will not be arrayed, which is also the case if this argument does not appear and is always true in electrical mode. If the scalar is nonzero, then the placement will use the current array parameters, as displayed in the **Cell Placement Control** pop-up, or set with the `PlaceSetArrayParams` function. If the argument is the name of an array, the array contains the arraying parameters. These parameters are:

array[0]	NX, integer number in the X direction.
array[1]	NY, integer number in the Y direction.
array[2]	DX, the real value spacing between cells in the X direction, in microns.
array[3]	DY, the real value spacing between cells in the Y direction, in microns.

The NX and NY values will be clipped to the range of 1 through 32767. The DX and DY are edge to adjacent edge spacing, i.e., when zero the elements will abut. If DX or DY is given the negative cell width or height, so that all elements appear at the same location, the corresponding NX or NY is taken as 1. Otherwise, there is no restriction on DX or DY.

If the boolean value *smash* is given and nonzero (TRUE), the cell will be flattened into the parent, rather than placed as an instance. The flatten-level is 1, so subcells of the cell (if any) become subcells of the parent. This argument is ignored if the cell being placed is a parameterized cell (pcell).

The *usegui* argument applies only when placing a pcell. If nonzero (TRUE), the **Parameters** panel will appear, and the function will block until the user dismisses the panel. The panel can be used to set cell parameters before instantiation. Initially, the parameters will be shown with default values, or values that were last given to `PlaceSetPCellParams`. If the *usegui* argument is not given or zero (FALSE), the default parameter set as updated with parameters given to `PlaceSetPCellParams` will be used to instantiate the cell immediately.

The final argument can be a null string or scalar 0 which is equivalent, an empty string, or a transform description in the format returned by `GetTransformString`. If null or not given, the arguemnt is ignored. In this case, the cell will be transformed before placement according to the current transform. Otherwise, the given transformation will be used when placing the instance. An empty string is taken as the identity transform. If the `UseTransform` mode is in effect, the current transform will be added to the string transform, giving an overall transfromation that will match geometry placement in this mode.

On success, the function returns 1, 0 otherwise.

(object_handle) `PlaceH(`*cellname*`,` *x*`,` *y* `[,` *refpt*`,` *array*`,` *smash*`,` *usegui*`,` *tfstring*`])`
This is similar to the `Place` function, however it returns a handle to the newly created instance. However, if the *smash* boolean is true or on error, a scalar 0 is returned.

(int) `PlaceSetArrayParams(`*nx*`,` *ny*`,` *dx*`,` *dy*`)`
This function provides array parameters which may be used when instantiating physical cells. These parameters will appear in the **Cell Placement Control** panel. The arguments are:

nx	Integer number in the X direction.
ny	Integer number in the Y direction.
dx	The real value spacing between cells in the X direction, in microns.
dy	The real value spacing between cells in the Y direction, in microns.

The *nx* and *ny* values will be clipped to the range of 1 through 32767. The *dx* and *dy* are edge to adjacent edge spacing, i.e., when zero the elements will abut. If *dx* or *dy* is given the negative cell width or height, so that all elements appear at the same location, the corresponding *nx* or *ny* is taken as 1. Otherwise, there is no restriction on *dx* or *dy*.

The function returns 1 and sets the array parameters in physical mode. In electrical mode, the function returns 0 and does nothing.

(int) `PlaceSetPCellParams(`*library*`,` *cell*`,` *view*`,` *params*`)`
This sets the default parameterized cell (pcell) parameters used when instantiating the pcell indicated by the *libname*/*cell*/*view*. If *library* is not given as a scalar 0, it is the name of the OpenAccess library containing the pcell super-master, whose name is given in the *cell* argument. The *view* argument can be passed a scalar 0 to indicate that the OpenAccess view name is "`layout`", or the actual view name can be passed if different. For *Xic* native pcells not stored in OpenAccess, the library and view should both be 0 (zero).

The *params* argument is a string providing the parameter values in the format of the pc_params property as applied to sub-masters and instances, i.e., values are constants and constraints are **not** included. Not all parameters need be given, only those with non-default values.

Be aware that there is no immediate constraint testing of the parameter values given to this function, though bad values will cause subsequent instantiation of the named pcell to fail. The `CheckPCellParams` fuction can be used to validate the params list before calling this function. When giving parameters for non-native pcells, it is recommended that the type specification prefixes be used, though an attempt is made internally to recognize and adapt to differing types.

The saved parameter set will be used for all instantiations of the pcell, until changed with another call to `PlaceSetPCellParams`. The placement is done with the `Place` script function, as for normal cells.

In graphical mode, the given parameter set will initialize the **Parameters** pop-up.

This function manages an internal table of cellname/parameter list associations. If 0 is given for all arguments, the table will be cleared. If the *params* argument is 0, the specified entry will be removed from the table. When the script terminates, parameter lists set with this function will revert to the pre-script values. Entries that were cleared by passing null arguments are **not** reverted, and remain cleared.

The function returns 1 on success, 0 if an error occurred, with an error message available from `GetError`.

(int) `Replace(`*cellname*`,` *add_xform*`,` *array*`)`

This will replace all selected subcells with *cellname*. The same transformation applied to the previous instance is applied to the replacing instance. In addition, if *add_xform* is nonzero, the current transform will be added. The function returns 1 if successful, 0 if the new cell could not be opened.

The *array* argument can be a scalar, or the name of an array containing four numbers. This argument specifies the arraying parameters for the instance placement, which apply in physical mode only. If a scalar 0 is passed, the placement will retain the same arraying parameters as the previous instance. If the scalar is nonzero, then the placement will use the current array parameters, as displayed in the **Cell Placement Control** pop-up, or set with the `PlaceSetArrayParams` function. If the argument is the name of an array, the array contains the arraying parameters. These parameters are:

array[0]	NX, integer number in the X direction.
array[1]	NY, integer number in the Y direction.
array[2]	DX, the real value spacing between cells in the X direction, in microns.
array[3]	DY, the real value spacing between cells in the Y direction, in microns.

The NX and NY values will be clipped to the range of 1 through 32767. The DX and DY are edge to adjacent edge spacing, i.e., when zero the elements will abut. If DX or DY is given the negative cell width or height, so that all elements appear at the same location, the corresponding NX or NY is taken as 1. Otherwise, there is no restriction on DX or DY.

(int) `OpenViaSubMaster(`*vianame*`,` *defnstr*`)`

This function will create if necessary and return the name of a standard via sub-master cell in memory. The first argument is the name of a standard via, as defined in the technology file or imported from Virtuoso. The second argument contains a string that specifies the parameters that differ from the default values. This can be null or empty if no non-default values are used. The format is the same as described for the `stdvia` property, with the standard via name token stripped (see 5.8.1).

On success, a name is returned. One can use this name with the `Place` function to instantiate the via. Otherwise, a fatal error is triggered.

F.6.2 Clipping Functions

(int) `ClipAround(`*object_handle1*`,` *all1*`,` *object_handle2*`,` *all2*`)`

This function will clip out the pieces of objects in the second handle list that intersect with objects in the first handle list.

If the boolean value *all1* is nonzero, all objects in the first handle are used for clipping, otherwise only the first object is used. If the boolean value *all2* is nonzero, all objects in the second handle list may be clipped, otherwise only the first object in the list is a candidate for clipping. Only boxes, polygons, and wires that appear in the second handle list will be clipped. The objects in the first handle list can be of any type, and labels and subcells will use the bounding box. The objects in the second list must be database objects, if they are are copies, no clipping is performed. The objects in the first list can be copies.

The newly created objects are added to the front of the second handle list, and the original object is removed from the list. The return value is the number of objects created, or -1 if either handle is empty or some other error occurred. The function fails if either handle does not reference an object list.

(object_handle) `ClipAroundCopy(`*object_handle1*`,` *all1* `,` *object_handle2*`,` *all2* `,` *lname*`)`
This function is similar to `ClipAround`, however no new objects are created in the database, and neither of the lists passed as arguments is altered. Instead, a new object list handle is returned, which references a list of "copies" of objects that are created by the clipping. The new objects are the pieces of the object or objects referenced by the second handle that do not intersect the object or objects referenced by the first handle.

If the boolean value *all1* is nonzero, all objects in the first handle are used for clipping, otherwise only the first object is used. If the boolean value *all2* is nonzero, all objects in the second handle list may be clipped, otherwise only the first object in the list is a candidate for clipping. Only boxes, polygons, and wires that appear in the second handle list will be clipped. The objects in the first handle list can be of any type, and labels and subcells will use the bounding box. The objects in the second list can be database objects or copies.

If *lname* is a non-empty string, it is taken as the name for a layer on which all of the returned objects will be placed. The layer will be created if it does not exist. If zero or an empty or null string is passed, the object copies will retain the layer of the original object from the second handle list.

The returned list can be used by most functions that expect a list of objects, however they are not copies of "real" objects. If no new object copy would be created by clipping, the function returns 0. The function will fail if either handle is not an object-list handle.

(int) `ClipTo(`*object_handle1*`,` *all1* `,` *object_handle2*`,` *all2*`)`
This function will clip objects referenced by the second handle to the boundaries of objects referenced by the first handle.

If the boolean value *all1* is nonzero, all objects in the first handle are used for clipping, otherwise only the first object is used. If the boolean value *all2* is nonzero, all objects in the second handle list may be clipped, otherwise only the first object in the list is a candidate for clipping. Only boxes, polygons, and wires that appear in the second handle list will be clipped. The objects in the first handle list can be of any type, and labels and subcells will use the bounding box. The objects in the second list must be database objects, if they are are copies, no clipping is performed. The objects in the first list can be copies.

The newly created objects are added to the front of the second handle list, and the original object is removed from the list. The return value is the number of objects created, or -1 if either handle is empty or some other error occurred. The function fails if either handle does not reference an object list.

(object_handle) `ClipToCopy(`*object_handle1*`,` *all1* `,` *object_handle2*`,` *all2* `,` *lname*`)`
This function is similar to `ClipTo`, however no new objects are created in the database, and neither of the lists passed as arguments is altered. Instead, a new object list handle is returned, which references a list of "copies" of objects that are created by the clipping. The new objects are the pieces of the object or objects referenced by the second handle that intersect the object or objects referenced by the first handle.

If the boolean value *all1* is nonzero, all objects in the first handle are used for clipping, otherwise only the first object is used. If the boolean value *all2* is nonzero, all objects in the second handle list may be clipped, otherwise only the first object in the list is a candidate for clipping. Only boxes, polygons, and wires that appear in the second handle list will be clipped. The objects in the first handle list can be of any type, and labels and subcells will use the bounding box. The objects in the second list can be database objects or copies.

If *lname* is a non-empty string, it is taken as the name for a layer on which all of the returned objects will be placed. The layer will be created if it does not exist. If zero or an empty or null string is passed, the object copies will retain the layer of the original object from the second handle list.

The returned list can be used by most functions that expect a list of objects, however they are not copies of "real" objects. If no new object copy would be created by clipping, the function returns 0. The function will fail if either handle is not an object-list handle.

(int) `ClipObjects(`*object_handle*`,` *merge*`)`

This function will clip boxes, polygons, and wires in the list on the same layer as the first such object in the list so that none of these objects overlap. Newly created objects are added to the front of the handle list, and deleted objects are removed from the list. Objects in the list that are not on the same layer as the first box, polygon, or wire or are not boxes, polygons or wires are ignored. If the merge argument is nonzero, adjacent new objects will be merged, otherwise the pieces will remain separate objects. If successful, the number of newly created objects is returned, otherwise -1 is returned. The function will fail if the handle does not reference an object list.

(object_handle) `ClipIntersectCopy(`*object_handle1*`,` *all1*`,` *object_handle2*`,` *all2*`,` *lname*`)`

This function returns a list of object copies which represent the exclusive-or of box, polygon, and wire objects in the two object lists passed. The lists are not altered in any way, and the new objects, being "copies", are not added to the database. Objects found in the lists that are not boxes, polygons, or wires are ignored. The new objects are placed on the layer with the name given in *lname*, which is created if it does not exist, independent of the originating layer of the objects. If a null string or 0 is passed for *lname*, the target layer is the first layer found in *object_handle1*, or *object_handle2* if *object_handle1* is empty. The *all1* and *all2* are integer arguments indicating whether to use only the first object in the list, or all objects in the list. If nonzero, then all boxes, polygons, and wires in the corresponding list will be used, otherwise only the first box, polygon, or wire will be processed. On success, a handle to a list of object copies is returned, zero is returned otherwise. A fatal error is triggered if either argument is not a handle to a list of objects.

F.6.3 Other Object Management Functions

(int) `ChangeLayer()`

This function will change the layer of all selected geometry to the current layer. This is similar to the functionality of the **Chg Layer** button in the **Modify Menu**.

(int) `Bloat(`*dimen*`,` *mode*`)`

Each selected object is bloated by the given dimension, similar to the **!bloat** command. The returned value is 0 on success, or 1 if there was a runtime error. This function will return 1 if not called in physical mode.

The second argument is an integer that specifies the algorithm to use for bloating. Giving zero specifies the default algorithm. See the description of the **!bloat** command (19.13.12) for documentation of the algorithms available.

(int) `Manhattanize(`*dimen*`,` *mode*`)`

Each selected non-Manhattan polygon or wire is converted to a Manhattan polygon or box approximation, similar to the **!manh** command. The first argument is a size in microns representing the smallest dimension of the boxes created to approximate the non-Manhattan parts. The second argument is a boolean value that specifies which of two algorithms to use. These algorithms are described with the **!manh** command.

The returned value is 0 on success, or 1 if there was a runtime error. This function will return 1 if not called in physical mode. The function will fail if the *dimen* argument is smaller than 0.01.

(int) `Join()`

The selected objects that touch or overlap are merged together into polygons, similar to the **!join** command. The returned value is 0 on success, 1 if there is a runtime error. This function will return 1 if not called in physical mode.

(int) `Decompose(`*vert*`)`

The selected polygons and wires are decomposed into elemental non-overlapping trapezoids (polygons) similar to the **!split** command. If the integer argument is nonzero, the decomposition favors a vertical orientation, otherwise the splitting favors horizontal. The returned value is 0 if called in physical mode, 1 if not called in physical mode (an error).

(int) `Box(`*left*`,` *bottom*`,` *right*`,` *top*`)`

The four arguments are real values specifying the coordinates of a rectangle in microns. Calling this function will generate a box on the current layer with the given coordinates. This provides functionality similar to the **box** menu button.

If the `UseTransform` function has been called to enable use of the current transform, the current transform will be applied to given coordinates before the box is created. The translation supplied to `UseTransform` is added to the coordinates before the current transform is applied.

The `Box` function will actually create a polygon if the current transform is being used and the rotation angle is 45 degrees or one of the other non-Manhattan angles.

(object_handle) `BoxH(`*left*`,` *bottom*`,` *right*`,` *top*`)`

This is similar to the `Box` function, but will return a handle to the new object. On error, a scalar 0 is returned.

(int) `Polygon(`*num*`,` *arraypts*`)`

This function creates a polygon on the current layer. The second argument is an array of values, taken as x-y pairs. The first pair of values must be the same as the last, i.e., the path must be closed. The first argument is the number of pairs of coordinates in the array. This provides functionality similar to the **polyg** menu button.

If the `UseTransform` function has been called to enable use of the current transform, the current transform will be applied to the given coordinates before the polygon is created. The translation supplied to `UseTransform` is added to the coordinates before the current transform is applied.

The `Polygon` function will actually create a box if the rotated figure can be so represented. The `Polygon` function will never create boxes unless use of the current transform is enabled.

(object_handle) `PolygonH(`*num*`,` *arraypts*`)`

This is similar to the `Polygon` function, but will return a handle to the new object. On error, a scalar 0 is returned.

(int) `Arc(`*x*`,` *y*`,` *rad1X*`,` *rad1Y*`,` *rad2X*`,` *rad2Y*`,` *ang_start*`,` *ang_end*`)`

This produces a circular or elliptical solid or ring-like figure, providing functionality similar to the **round**, **donut**, and **arc** buttons in the physical side menu.

x, y	center coordinates
rad1X, rad1Y	x and y inner radii
rad2X, rad2Y	x and y outer radii
ang_start	starting angle in degrees
ang_end	ending angle in degrees

All dimensions are given in microns. The first two arguments provide the center coordinates. The second two arguments are the inner radius in the X and Y directions. If these differ, the inner radus will be elliptical, otherwise it will be circular. If both are zero, the figure will not have an inner surface.

Similarly, the next two arguments specify the outer radius, X and Y directions separately. Both are required to be larger than the inner radius counterpart.

The final two arguments are the start and end angle, given in degrees. If *ang_start* and *ang_end* are equal, a donut (ring figure) is produced. If the outer and inner radii are equal, a solid figure is produced. Angles are defined from the positive x-axis, in a counter-clockwise sense. The arc is generated in a clockwise direction.

If the `UseTransform` function has been called to enable use of the current transform, the current transform will be applied to the arc coordinates before the arc is created. The translation supplied to `UseTransform` is added to the coordinates before the current transform is applied.

The function returns 1 on success, 0 otherwise.

(object_handle) `ArcH(`*x*`,` *y*`,` *rad1X*`,` *rad1Y*`,` *rad2X*`,` *rad2Y*`,` *ang_start*`,` *ang_end*`)`
This is similar to the `Arc` function, but will return a handle to the new object. On error, a scalar 0 is returned.

(int) `Round(`*x*`,` *y*`,` *rad*`)`
This a simplification of the `Arc` function which simply creates a circular disk object at the location specified in the first two arguments. All dimensions are in microns. The third argument specifies the radius.

The function returns 1 on success, 0 otherwise.

(object_handle) `RoundH(`*x*`,` *y*`,` *rad*`)`
This is similar to the `Round` function, but will return a handle to the new object. On error, a scalar 0 is returned.

(int) `HalfRound(`*x*`,` *y*`,` *rad*`,` *dir*`)`
This is a simplification of the `Arc` function which creates a half-circular figure. The first two arguments indicate the center of an equivalent full circle, i.e., it is the midpoint of the flat edge. The *dir* argument is an integer 0–7 which specifies the orientation, in increments of 45 degrees. With 0, the flat section is horizontal with the curved surface on top. The dir rotates clockwise, so that a value of 2 would produce a figure that looks like the letter D.

The function returns 1 on success, 0 otherwise.

(object_handle) `HalfRoundH(`*x*`,` *y*`,` *rad*`,` *dir*`)`
This is similar to the `HalfRound` function, but will return a handle to the new object. On error, a scalar 0 is returned.

(int) `Sides(`*numsides*`)`
This sets the number of segments to use in generating round objects, for the current display mode (electrical or physical). The function returns the present value for this parameter. This is similar to the **sides** side menu button in physical mode. It simply sets the RoundFlashSides variable, or clears the variable if the number of sides given is the default. Similarly, in electrical mode it is

similar to the **sides** entry in the menu from the **shape** button in the side menu, and sets or clears the ElecRoundFlashSides variable.

(int) `Wire(`*width*`,` *num*`,` *arraypts*`,` *end_style*`)`

This function creates a wire on the current layer. The first argument is the width of the wire in microns. The third argument is the name of an array of coordinates, taken as x-y pairs. The second argument is the number of coordinate pairs in the array. The fourth argument is 0, 1, or 2 to set the end style to flush, rounded, or extended, respectively. This provides the functionality of the **wire** menu button.

If the `UseTransform` function has been called to enable use of the current transform, the current transform will be applied to the given coordinates before the wire is created. The translation supplied to `UseTransform` is added to the coordinates before the current transform is applied. The variable NoWireWidthMag will suppress changes to the wire width due to the magnification component of the current transform when set.

(object_handle) `WireH(`*width*`,` *num*`,` *arraypts*`,` *end_style*`)`

This is similar to the `Wire` function, but will return a handle to the new object. On error, a scalar 0 is returned.

(int) `Label(`*text*`,` *x*`,` *y* `[,` *width*`,` *height*`,` *flags*`)]`

This function creates a label on the current layer. The function takes a variable number of arguments, but the first three must be present. The first argument is of string type and contains the label text. The next two arguments specify the x and y coordinates of the label reference point.

The remaining arguments are optional. The *width* and *height* specify the size of the bounding box into which the text will be rendered, in microns. if both are zero or negative or not given, a default size will be used. If only one is given a value greater than zero, the other will be computed using a default aspect ratio. If both are greater than zero, the text will be squeezed or stretched to conform.

The *flags* argument is a label flags word used in *Xic* to set various label attributes, as described in C.2. If given, the `Justify` function and `UseTransform` function settings will be ignored, and these attributes will be set from the *flags*. If *flags* is not given, the functions will set the justification and transformation.

This function always returns 1.

(object_handle) `LabelH(`*text*`,` *x*`,` *y* `[,` *width*`,` *height*`,` *xform*`)]`

This is similar to the `Label` function, but will return a handle to the new object. On error, a scalar 0 is returned.

(int) `Logo(`*string*`,` *x*`,` *y* `[,` *width*`,` *height*`])`

This creates and places physical text, i.e., text that is constructed with database polygons that will appear in the mask layout. The function takes a variable number of arguments, but the first three must be present. The first argument is of string type and contains the label text. The next two arguments specify the x and y coordinates of the reference point, which is dependent on the current justification, as set with the `Justify` function. The default is the lower-left corner of the bounding box. The text will be transformed according to the current transform.

The remaining arguments are optional. The *width* and *height* specify the approximate size of the rendered text. Unlike the `Label` function, the text aspect ratio is fixed. The first of *height* or *width* which is positive will be used to set the "pixel" size used to render the text, by dividing this value by the character cell height or width of the default font. Thus, the rendered text size will only be accurate for this font, and will scale with the number of pixels used in the "pretty" fonts. One must experiment with a chosen font to obtain accurate sizing. If neither parameter is given and positive, a default size will be used.

This provides the functionality of the **logo** menu button, and is sensitive to the following variables.

LogoEndStyle
LogoPathWidth
LogoAltFont
LogoPrettyFont
LogoPixelSize
LogoToFile

This function always returns 1.

(int) `Justify(`*hj*`, `*vj*`)`
This sets the justification for text created with the **logo** and **label** commands and corresponding script functions. The arguments can have the following values:

hj/vj	horizontal	vertical
0	left	bottom
1	center	center
2	right	top

Values out of range will preserve the present justification setting. The function always returns 1.

(int) `Delete()`
This function deletes all selected objects from the database.

(int) `Erase(`*left*`, `*bottom*`, `*right*`, `*top*`)`
This function erases the rectangular area defined by the arguments. Polygons, wires, and boxes are appropriately clipped. The erase function has no effect on subcells or labels. This provides an erase capability similar to the **erase** menu button.

(int) `EraseUnder()`
This function will erase geometry from unselected objects that intersect with objects that are selected. This is equivalent to the **Erase Under** command in *Xic*. This function always returns 1.

(int) `Yank(`*left*`, `*bottom*`, `*right*`, `*top*`)`
This function puts the geometry in the specified rectangle in yank buffer 0. It can be placed with the `Put` function, or the **put** command. This provides a yank capability similar to the **erase** button in the side menu.

(int) `Put(`x`, `y`, `*bufnum*`)`
This puts the contents of the indicated yank buffer in the current layout, with the lower left at x, y. The *bufnum* is the yank buffer index, which can be 0–4. Buffer 0 is the most recent yank or erase, buffer 1 is the next most recent, etc. This provides functionality similar to the **put** button in the side menu.

(int) `Xor(`*left*`, `*bottom*`, `*right*`, `*top*`)`
This function exclusive-or's the rectangular area defined by the arguments with boxes, polygons, and wires on the current layer. Existing objects become clear areas. This provides functionality similar to the **xor** button in the side menu.

(int) `Copy(`*fromx*`, `*fromy*`, `*tox*`, `*toy*`, `*repcnt*`)`
Copies of selected objects are created and placed such that the point specified by the first two arguments is moved to the location specified by the second two arguments.

The *repcnt* is an integer replication count in the range 1–100000, which will be silently taken as one if out of range. If not one, multiple copies are made, at multiples of the translation factors given.

This provides functionality similar to the **Copy** button in the **Modify Menu**. The return value is 1 if there were no errors and something was copied, 0 otherwise.

(int) `CopyToLayer(`*fromx*`,` *fromy*`,` *tox*`,` *toy*`,` *oldlayer*`,` *newlayer*`,` *repcnt*`)`
This is similar to the `Copy` function, but allows layer change. If *newlayer* is 0, null, or empty, *oldlayer* is ignored and the function behaves identically to `Copy`. Otherwise the *newlayer* string must be a layer name. If *oldlayer* is 0, null, or empty, all copied objects are placed on *newlayer*. Otherwise, *oldlayer* must be a layer name, in which case only objects on *oldlayer* will be placed on *newlayer*, other objects will remain on the same layer. Subcell objects are copied as in `Copy`, i.e., the layer arguments are ignored.

(int) `Move(`*fromx*`,` *fromy*`,` *tox*`,` *toy*`)`
This function moves the selected objects such that the reference point specified in the first two arguments is moved to the point specified by the second two arguments. This provides functionality similar to the **Move** button in the **Modify Menu**. The return value is 1 if there were no errors and something was moved, 0 otherwise.

(int) `MoveToLayer(`*fromx*`,` *fromy*`,` *tox*`,` *toy*`,` *oldlayer*`,` *newlayer*`)`
This is similar to the `Move` function, but allows layer change. If *newlayer* is 0, null, or empty, *oldlayer* is ignored and the function behaves identically to `Move`. Otherwise the *newlayer* string must be a layer name. If *oldlayer* is 0, null, or empty, all moved objects are placed on *newlayer*. Otherwise, *oldlayer* must be a layer name, in which case only objects on *oldlayer* will be placed on *newlayer*, other objects will remain on the same layer. Subcell objects are moved as in `Move`, i.e., the layer arguments are ignored.

(int) `Rotate(`*x*`,` *y*`,` *ang*`,` *remove*`)`
The selected objects are rotated counter-clockwise by *ang* (in degrees) about he point specified in the first two arguments. This provides functionality similar to the **spin** button in the side menu.

If the boolean argument *remove* is true (nonzero), the original objects will be deleted. Otherwise, the original objects are retained, and will become deselected.

The return value is 1 if there were no errors and something was rotated, 0 otherwise.

Note: in releases prior to 3.0.5, the *remove* argument was absent and effectively 0 in the current function implementation.

(int) `RotateToLayer(`*x*`,` *y*`,` *ang*`,` *oldlayer*`,` *newlayer*`,` *remove*`)`
This is similar to the `Rotate` function, but allows layer change. If *newlayer* is 0, null, or empty, *oldlayer* is ignored and the function behaves identically to `Rotate`. Otherwise the *newlayer* string must be a layer name. If *oldlayer* is 0, null, or empty, all rotated objects are placed on *newlayer*. Otherwise, *oldlayer* must be a layer name, in which case only objects on *oldlayer* will be placed on *newlayer*, other objects will remain on the same layer. Subcell objects are rotated as in `Rotate`, i.e., the layer arguments are ignored.

If the boolean argument *remove* is true (nonzero), the original objects will be deleted. Otherwise, the original objects are retained, and will become deselected.

The return value is 1 if there were no errors and something was rotated, 0 otherwise.

Note: in releases prior to 3.0.5, the *remove* argument was absent and effectively 0 in the current function implementation.

(int) `Split(`*x*`,` *y*`,` *flag*`,` *orient*`)`
This will sever selected objects along a vertical or horizontal line through x, y if *flag* is nonzero. If *orient* is 0, the break line is vertical, otherwise it is horizontal. If *flag* is zero, the function will return 1 if an object would be split, 0 otherwise, though no objects are actually split. This provides functionality similar to the **break** button in the side menu.

(int) `Flatten(`*depth*`,` *use_merge*`,` *fast_mode*`)`

The selected subcells are flattened into the current cell, recursively to the given depth, similar to the effect of the **Flatten** button in the **Edit Menu**.

The *depth* argument may be an integer representing the depth into the hierarchy to flatten: 0 for top-level subcells only, 1 to include second-level subcells, etc. This argument can also be a string starting with '`a`' to signify flattening all levels. A negative depth also signifies flattening all levels.

The *use_merge* argument is a boolean which if nonzero indicates that new objects will be merged with existing objects when added to the current cell. This is the same merging as specified in the **Editing Setup** panel from the **Edit Menu**, or corresponding variables.

If the boolean argument *fast_mode* is nonzero, "fast" mode is used, meaning that there will be no undo list generation and no object merging. This is not undoable so should be used with care.

The function returns 1 on success, 0 otherwise, with an error message probably available from `GetError`.

`Layer(`*string*`,` *mode*`,` *depth*`,` *recurse*`,` *noclear*`,` *use_merge*`,` *fast_mode*`)`

This is very similar to the **!layer** command, and operations from the **Evaluate Layer Expression** panel brought up with the **Layer Expression** button in the **Edit Menu**. The *string* is of the form

"*new_layer_name* [=] *layer_expression*".

The *mode* argument is an integer which sets the split/join mode, similar to the keywords in the **!layer** command, and the buttons in the **Evaluate Layer Expression** panel. Only the two least-significant bits of the integer value are used.

0	default
1	horizontal split
2	vertical split
3	join

The *depth* is the search depth, which can be an integer which sets the maximum depth to search (0 means search the current cell only, 1 means search the current cell plus the subcells, etc., and a negative integer sets the depth to search the entire hierarchy). This argument can also be a string starting with '`a`' such as "`a`" or "`all`" which specifies to search the entire hierarchy.

The *recurse* argument is a boolean value which corresponds to the "`-r`" option of the **!layer** command, or the **Recursively create in subcells** check box in the **Evaluate Layer Expression** panel. If nonzero, evaluation will be performed in subcells to depth, using only that cell's geometry. When zero, geometry is created in the current cell only, using geometry found in subcells to depth.

If the boolean argument *noclear* is true, the target layer will not be cleared before expression evaluation. This corresponds to the "`-c`" option of the **!layer** command, and the **Don't clear layer before evaluation** button in the **Evaluate Layer Expression** panel.

The boolean argument *use_merge* corresponds to the "`-m`" option in the **!layer** command, and the **Use object merging while processing** check box in the **Evaluate Layer Expression** panel. When nonzero, new objects will be merged with existing objects when added to a cell.

The *fast_mode* argument is a boolean value that corresponds to the "`-f`" option in the **!layer** command, and the **Fast mode** check box in the **Evaluate Layer Expression** panel. When nonzero, undo list processing and merging are skipped for speed and to reduce memory use. However, the result is not undoable so this flag should be used with care.

There is no return value; the function either succeeds or will terminate the script on error.

F.6.4 Property Management

The functions described in this section provide an interface for working with properties.

When specifying the property "number" for electrical mode properties, either a number or string equivalent can be used. The string equivalent is a prefix of one of the supported property names. In addition, some of the properties have a letter that any word that starts with the letter will indicate that property. The idea was that each property could be keyed by a single letter, and this is almost still true (node is the exception).

The following table identifies the recognized strings. Not all of these properties apply in all functions. The listed order is the order of testing, the first match yields the equivalence.

Number	Name	String
1	model	prefix
2	value	prefix
3	param	prefix
3	initc	prefix
4	other	prefix
11	name	prefix
5	nophys	prefix or starts with 'y' or 'Y'
6	virtual	prefix or starts with 't' ot 'T'
7	flatten	prefix
8	range	prefix
10	node	prefix
18	nosymb	prefix or starts with 's' or 'S'
20	macro	prefix or starts with 'c' or 'C'
21	devref	prefix

The initc is an archaic alias for the param property that is still recognized. In some functions, an additonal keyword "`all`" is recognized in a way that has significance to the function. If the string does not match, an error is indicated.

(prpty_handle) `PrpHandle(`*object_handle*`)`
: This function returns a handle to the list of properties of the object referenced by the passed object handle. The function fails if the argument is not a valid object handle, use `CellPrpHandle` to list cell properties.

(prpty_handle) `GetPrpHandle(`*number*`)`
: Since there can be arbitrarily many properties defined with the same number, a generator function is used to read properties one at a time. This function returns a handle to a list of the properties that match the *number* passed. This applies to the first object in the selection queue (the most recent object selected). The returned value is used by other functions to actually retrieve the property text.

  If the *number* argument is a prefix of "`all`", then any property string will be returned. In physical mode, the *number* argument should otherwise be an integer. In electrical mode, the *number* argument can have string form as described in the introduction to this section.

(prpty_handle) `CellPrpHandle()`
: This function returns a handle to the list of properties of the current cell, applicable to the current display mode in the main window.

(prpty_handle) `GetCellPrpHandle(`*number*`)`

Since there can be arbitrarily many properties defined with the same number, a generator function is used to read properties one at a time. This function returns a handle to a list of the properties that match the *number* passed, from the current cell. The returned value is used by other functions to actually retrieve the property text.

A prefix of the string "`all`" can be passed for the *number* argument, in which case the handle will reference all properties of the cell. In physical mode, the *number* argument should otherwise be an integer. In electrical mode, the *number* argument can have string form as described in the introduction to this section.

(int) `PrpNext(`*prpty_handle*`)`
This function causes the referenced property of the passed handle to be advanced to the next in the list. If there are no other properties in the list, the handle is closed, and 0 is returned. Otherwise, the handle (same as the argument) is returned. The number of remaining reference objects can be obtained with the `HandleContent` function.

(int) `PrpNumber(`*prpty_handle*`)`
This function returns the number of the property referenced by the handle.

(string) `PrpString(`*prpty_handle*`)`
This function returns the string of the property referenced by the handle. The "raw" string is returned, meaning that if the property comes from an electrical object, all of the detail from the internal property string is returned.

(string) `PrptyString(`*obj_or_prp_handle*`,` *number*`)`
The first argument can be a property handle, or an object handle. If a property handle is given, the function returns the string of the first property referenced by the handle that matches the *number*. If the *number* argument is a prefix of "`all`", then any property string will be returned. In physical mode, the *number* argument should otherwise be an integer. In electrical mode, the *number* argument can be a string, as described in the introduction to this section. The handle is set to reference the next property in the reference list, following the one returned. When there are no more properties, this function returns a null string.

If the first argument is an object handle, the function returns the strings from properties or pseudo-properties for the object referenced by the handle.

In physical mode, the function will locate a property with the given number, and return its string. If no property is found with that number, and a pseudo-property for the object matches the number, then the pseudo-property string is returned. If no matching pseudo-property is found, a null string is returned. Note: objects can be modified through setting pseudo-properties using the `PrptyAdd` function.

In electrical mode, the number argument can be a string, as described in the introduction to this section. In the case of an object handle, the "`all`" keyword is not supported.

The function will fail if the argument is not a valid object or property handle. Use `GetCellPropertyString` to obtain strings from cell properties.

If the requested property is a `name` property of an electrical device or subcircuit, only the name is returned (the internal property string is more complex). Otherwise the "raw" string is returned.

(string) `GetPropertyString(`*number*`)`
This function searches the selection queue for an object with a property matching *number*. The string for the first such property found is returned. A null string is returned if no matching property was found.

(string) `GetCellPropertyString(`*number*`)`

This function searches the properties of the current cell, and returns the string for the first property found that matches *number*. If no match, a null string is returned.

(int) `PrptyAdd(`*object_handle*`,` *number*`,` *string*`)`

This function will create a new property using the *number* and *string* provided, on the object referenced by the handle. The object must be defined in the current cell. The function will fail if the handle is invalid. Use `CellPropertyAdd` to add properties to the current cell.

In physical mode, the property number can take any non-negative value. This includes property numbers that are used by *Xic* for various purposes in the range 7000–7199. Unless the user is expecting the *Xic* interpretation of the property number, these numbers should be avoided. It is the caller's responsibility to ensure that the properties in this range are applied to the appropriate objects, in the correct context and with correct syntax, as there is little or no checking. Adding some properties in this range such as flags, flatten, or a pcell property will automatically remove an existing property with the same number, if any.

The pseudo-properties in the range 7200–7299 will have their documented effect when applied, and no property is added (see 10.1.2),

In electrical mode, it is possible to set these properties of device instances:

name, model, value, param, devref, other, range nophys, symblc

and the following properties of subcircuit instances:

name, param, other, flatten, range nophys, symblc.

Attempts to set properties not listed here will silently fail. The object must be defined in the current cell, thus the mode must be electrical.

If the function succeeds, 1 is returned. otherwise 0 is returned.

(int) `AddProperty(`*number*`,` *string*`)`

This function adds a property with the given number and string to all selected objects.

In physical mode, the property number can take any non-negative value. This includes property numbers that are used by *Xic* for various purposes in the range 7000–7199. Unless the user is expecting the *Xic* interpretation of the property number, these numbers should be avoided. It is the caller's responsibility to ensure that the properties in this range are applied to the appropriate objects, in the correct context and with correct syntax, as there is little or no checking.

The pseudo-properties in the range 7200–7299 will have their documented effect when applied, and no property is added,

In electrical mode, it is possible to set these properties of device instances:

name, model, value, param, devref, other, range nophys, symblc

and the following properties of subcircuit instances:

name, param, other, flatten, range nophys, symblc.

Attempts to set properties not listed here will silently fail. The object must be defined in the current cell, thus the mode must be electrical.

The number of properties added plus the number of pseudo-properties applied is returned.

(int) `AddCellProperty(`*number*`,` *string*`)`

This function adds a property to the current cell.

In physical mode, the property number can take any non-negative value. This includes property numbers that are used by *Xic* for various purposes in the range 7000–7199. Unless the user is expecting the *Xic* interpretation of the property number, these numbers should be avoided. It is the caller's responsibility to ensure that the properties in this range are applied to the appropriate objects, in the correct context and with correct syntax, as there is little or no checking. Adding some properties in this range such as flags, flatten, or a pcell property will automatically remove an existing property with the same number, if any.

Numbers in the pseudo-property range 7200–7299 will do nothing.

In electrical mode, it is possible to set the param, other, virtual, flatten, macro, node, name, and symbolic properties of the current cell. The last three are not "user settable" but are needed when building up a new circuit cell in memory, as in the scripts produced by the **!mkscript** command. The string should have the format as read from a native cell file.

The function returns 1 if the operation was successful, 0 otherwise.

(int) `PrptyRemove(`*object_handle*`,` *number*`,` *string*`)`

This function will remove properties matching the given *number* and *string* from the object referenced by the handle.

In physical mode, the property number can take any non-negative value. This includes property numbers that are used by *Xic* for various purposes in the range 7000–7199. It is the caller's responsibility to make sure that removal of properties in this range is appropriate. Giving numbers in the pseudo-property range 7200–7299 will do nothing.

If the *string* is null or empty, only the *number* is used for comparison, and all properties with that number will be removed. Otherwise, if the *string* is a prefix of the property string and the numbers match, the property will be removed.

In electrical mode, it is possible to remove these properties of device instances:

name, model, value, param, devref, other, range nophys, symblc

and the following properties of subcircuit instances:

name, param, other, flatten, range nophys, symblc.

Attempts to remove properties not listed here will silently fail. Except for other, the string argument is ignored. For other properties, the string is used as above to identify the property to delete.

Objects must be defined in the current cell. The function returns the number of properties removed.

(int) `RemoveProperty(`*number*`,` *string*`)`

This function will remove properties from selected objects.

In physical mode, the property number can take any non-negative value. This includes property numbers that are used by *Xic* for various purposes in the range 7000–7199. It is the caller's responsibility to make sure that removal of properties in this range is appropriate. Giving numbers in the pseudo-property range 7200–7299 will do nothing.

If the *string* is null or empty, only the *number* is used for comparison, and all properties with that number will be removed. Otherwise, if the *string* is a prefix of the property string and the numbers match, the property will be removed.

In electrical mode, it is possible to remove these properties of device instances:

name, model, value, param, devref, other, range nophys, symblc and the following properties of subcircuit instances:

name, param, other, flatten, range nophys, symblc.

Attempts to remove properties not listed here will silently fail. Except for other, the string argument is ignored. For other properties, the string is used as above to identify the property to delete.

The number of properties removed is returned.

(int) `RemoveCellProperty(`*number*`,` *string*`)`
This function will remove properties from the current cell.

In physical mode, the property number can take any non-negative value. This includes property numbers that are used by *Xic* for various purposes in the range 7000–7199. It is the caller's responsibility to make sure that removal of properties in this range is appropriate. Giving numbers in the pseudo-property range 7200–7299 will do nothing.

If the *string* is null or empty, only the *number* is used for comparison, and all properties with that number will be removed. Otherwise, if the *string* is a prefix of the property string and the numbers match, the property will be removed.

In electrical mode, it is possible to remove the param, other, virtual, flatten, and macro properties of the current cell. Except for other, the string argument is ignored. For other properties, the string is used as above to identify the property to delete.

The function returns the number of properties removed.

F.7 Computational Geometry and Layer Expressions

F.7.1 Trapezoid Lists and Layer Expressions

For the functions described below, a "zoidlist" argument can actually have the following data types:

zoidlist	Obviously
integer zero	Implies an empty zoidlist
integer nonzero	Implies the reference zoidlist
string	The string is parsed as a layer expression, which is evaluated, and the result used
layer_expr	evaluate layer expression, use result

(int) `SetZref(`*arg*`)`
This function sets the reference zoidlist. The reference zoidlist represents the current "background" needed by some functions and operators which manipulate zoidlists. For example, when a zoidlist is polarity inverted, the reference zoidlist specifies the boundary of the inversion, i.e., the inverse of an empty zoidlist would be the reference zoidlist.

The reference zoidlist can be set from various types of object passed as the *arg*. This can be a zoidlist, or an object handle, or an array of size 4 or larger, which contains rectangle coordinates in microns in order left, bottom, right, top. The argument can also be the constant 0, in which case the reference zoid list will be the boundary of the physical current cell, or a large "infinity" box if there is no current cell. This is the default if no reference zoid list is given.

This function will return 1 and fails only if the argument is not an appropriate type.

(zoidlist) `GetZref()`
This function returns the current reference zoidlist, which will be empty if no reference area has been set with `SetZref` or otherwise.

(int) `GetZrefBB(`*array*`)`
This will return the bounding box of the reference zoidlist, as returned from `GetZref`. If the reference zoidlist is empty, the bounding box of the current cell is returned. The coordinates are in microns, in order left, bottom, right, top. On success, the function returns 1. If there is no reference zoidlist or current cell, 0 is returned.

(int) `AdvanceZref(clear,` *array*`)`
This function allows iteration over a given area by establishing a grid over the area and incrementally setting the reference area (see `SetZref`) to elements of the grid. The grid is aligned from the lower-left corner of the given area and iteration advances right and up. The reference area is set to the intersection of the grid element area and the given area. The size of the square grid elements is given by the PartitionSize variable, or defaults to 100 microns if this variable is not set.

The second argument is an array of size 4 or larger, or 0. If 0, the given area is taken to be the bounding box of the current cell. Otherwise, the array elements define the given rectangular area, in microns, in order left, bottom, right, top.

With the boolean first argument set to zero, the function will set the reference area to the first (lower left) or next grid element intersection area and return 1. The function will return zero when it advances past the last grid element that overlaps the given area, at which time the reference area is returned to the default value. Thus, this function can be used in a loop to limit the computation area for each iteration, for large cells that would be inefficient to process in one step.

If the first argument is nonzero, the internal state is cleared. This should be called if the iteration is not complete and one wishes to start a new loop.

(zoidlist) `Zhead(`*zoidlist*`)`
This function will remove the first trapezoid from the passed trapezoid list, and return it as a new list. If the passed list is empty, the returned list will be empty. If the passed list contains a single trapezoid, it will become empty.

(int) `Zvalues(`*zoidlist*`,` *array*`)`
This function will return the coordinates of the first trapezoid in the list in the array, which must have size 6 or larger. The order of the values is

0	x lower-left
1	x lower-right
2	y lower
3	x upper-left
4	x upper-right
5	y upper

On success, 1 is returned. If the passed trapezoid list is empty, the return value is 0 and the array is untouched.

(int) `Zlength(`*zoidlist*`)`
This function returns the number of trapezoids contained in the list passed as an argument.

(int) `Zarea(`*zoidlist*`)`
This function returns the total area of the trapezoids contained in the list passed as an argument, in square microns. This does not account for overlapping trapezoids, call `GeomOr` first if overlapping trapezoids are present (lists returned from the script functions have already been clipped/merged unless otherwise noted).

(zoidlist) `GetZlist(`*layersrc*`,` *depth*`)`

This function returns a zoidlist from the layer source given in the first argument, which is a string in the form

lname[`.`*stname*][`.`*cellname*]

Any of *lname*, *stname*, *cname* can be double-quoted, which must be true if the token contains the separation char '`.`'. The *stname* is the name of a symbol table, the *cname* is tha name of a cell found in the symbol table. If there are only two fields, the second field is *cname*, and the current symbol table is understood. If no *cname* is given, the current cell is understood.

The returned list is clipped to the current reference area (see `SetZref`). The second argument is the hierarchy depth to search, which can be a non-negative integer or a string starting with '`a`' to indicate "`all`". If not called in physical mode, an empty list is returned.

The layer specification can also be given in the form

lname`.@`*dbname*

where *dbname* is the name of a saved database. Operation will be similar to the `GetZlistDb` script function.

(zoidlist) `GetSqZlist(`*layername*`)`

This function returns a trapezoid list derived from objects in the selection queue on the layer whose name is passed as the argument. Labels are ignored, as are subcells unless the layer name is the special name "$$", in which case the subcell bounding boxes are returned.

This function can be called successfully only in physical mode.

(zoidlist) `TransformZ(`*zoidlist*`,` *refx*`,` *refy*`,` *newx*`,` *newy*`)`

Return a transformed copy of the passed trapezoid list. The transform should have been set previously with `SetTransform` or equivalent. The original list is not touched and can be closed if no longer needed. The function internally converts each input trapezoid to a polygon, applies the transformation to the polygon coordinates, then decomposes the polygons into a new trapezoid list, which is returned.

The remaining arguments are "reference" and "new" coordinates, which provide for translations. The reference point is the point about which rotations and mirroring are performed, and is translated to the new location, if different.

(zoidlist) `BloatZ(`*dimen*`,` *zoidlist*`,` *mode*`)`

This function returns a new zoidlist which is a bloated version of the zoidlist passed as an argument (similar to the **!bloat** command). Edges will be pushed outward or pulled inward by *dimen* (positive values push outward). The *dimen* is given in microns.

The third argument is an integer that specifies the algorithm to use for bloating. Giving zero specifies the default algorithm. See the description of the **!bloat** command (19.13.12) for documentation of the algorithms available.

(zoidlist) `ExtentZ(`*zoidlist*`)`

This will return a zoidlist with at most one component: a rectangle giving the bounding box of the list given as an argument. If the passed list is null, the return is a null list.

(zoidlist) `EdgesZ(`*dimen*`,` *zoidlist*`,` *mode*`)`

This returns a list of zoids that in some way describe edges in the zoid list passed. The *dimen* is given in microns.

The *mode* is an integer which specifies the algorithm to use to define the edges. The values 0–3 are equivalent to the `BloatZ` function returning edges only, with the four corner fill-in modes.

mode 0
Provides an edge template as from the `BloatZ` function with corner fill-in mode 0 (rounded corners).

mode 1
Provides an edge template as from the `BloatZ` function with corner fill-in mode 1 (flat corners).

mode 2
Provides an edge template as from the `BloatZ` function with corner fill-in mode 2 (projected corners).

mode 3
Provides an edge template as from the `BloatZ` function with corner fill-in mode 3 (no corner fill).

mode 4
The zoid list is logically merged into distinct polygons, and a "halo" extending outside of the polygon by width *dimen* (positive value taken) is constructed. The trapezoids describing the halo are returned.

mode 5
The zoid list is logically merged into distinct polygons, and a wire object is constructed using each polygon vertex list. The wire width is twice the *dimen* value passed. The trapezoid list representing the wire area is returned. This may fail and give strange shapes if the dimensions of a polygon are smaller than half the wire width.

mode 6
For each zoid in the *zoidlist* argument, a new zoid is constructed from each edge that covers the area within +/- *dimen* normal to the edge. The list of new zoids is returned.

(zoidlist) `ManhattanizeZ(`*dimen*`,` *zoidlist*`,` *mode*`)`

This function returns a new zoidlist which is a Manhattan approximation of the zoidlist passed as an argument (similar to the **!manh** command). The first argument is the minimum rectangle width or height in microns used to approximate non-Manhattan pieces. The third argument is a boolean which specifies which of the two algorithms to employ. These algorithms are described with the **!manh** command, though in this function there is no reassembly into polygons.

All of the returned trapezoids are rectangles. The function will fail if the argument is smaller than 0.01.

(zoidlist) `RepartitionZ(`*zoidlist*`)`

This is a rather obscure function that conditions a list of trapezoids so that the area covered will be constructed with trapezoids that are as long (horizontally) as possible. Logically, this is what would happen if the initial trapezoid list was converted to distinct polygons, then split back into trapezoids.

(zoidlist) `BoxZ(`*l*`,` *b*`,` *r*`,` *t*`)`

This function returns a zoidlist containing a single trapezoid which represents the box given in the arguments. The given coordinates are in microns. This function never fails.

(zoidlist) `ZoidZ(`*xll*`,` *xlr*`,` *yl*`,` *xul*`,` *xur*`,` *yu*`)`

This function returns a zoidlist containing a single horizontal trapezoid which represents the horizontal trapezoid given in the arguments. The six numbers must represent a non-degenerate figure or the function will fail. The given coordinates are in microns.

(zoidlist) `ObjectZ(`*object_handle all*`)`

This function returns a zoidlist which is generated by fracturing the outlines of the objects in the *object_handle*. If *all* is 0, only the first object in the list is used. If *all* is nonzero, all objects in the list are used. This function will fail if the first argument is not a handle to an object list.

(layer_expr) `ParseLayerExpr(`*string*`)`
This function returns a variable which contains a parse tree for a layer expression contained in the string passed as an argument. The resulting variable is used to rapidly evaluate the layer expression. The return value can not be assigned or otherwise manipulated, and can only be passed to functions that expect this variable type. The function will fail on a parse error in the layer expression.

(zoidlist) `EvalLayerExpr(`*layer_expr*`, `*zoidlist*`, `*depth*`, `*isclear*`)`
This function evaluates the layer expression passed as the first argument. The first argument can be a string containing the layer expression, or a return from `ParseLayerExpr`. If the second argument is nonzero, it is taken as a reference zoidlist. If 0, the current reference zoidlist (as set with `SetZref`) will be used. The third argument is the depth into the cell hierarchy to process. This can be an integer, with 0 representing the current cell only, or a string starting with '`a`' to indicate use of all levels of the hierarchy. If *isclear* is 0, the returned zoidlist will represent all areas within the reference where the layer expression is "true". if *isclear* is nonzero, the complement regions will be returned. The function will fail on a parse or evaluation error.

(int) `TestCoverageFull(`*layer_expr*`, `*zoidlist*`, `*minsize*`)`
This function will return an integer value indicating the coverage of the layer expression given in the first argument over the regions described in the second argument. The first argument can be a string containing a layer expression, or a return from `ParseLayerExpression`. If the second argument is 0, the current reference zoidlist as set with `SetZref` is assumed. This defaults to tha area of the current cell.

The third argument is an integer which gives the minimum dimension in internal units of trapezoids which will be considered in the result. Sub-dimensional trapezoids are ignored. This minimizes false-positive tests due to "slivers" caused by clipping errors in non-Manhattan geometry. If the geomentry is known to be Manhattan, 0 can be used. If 45's only, 2 is recommended, otherwise 4. Negative values are taken as zero.

The function tests each dark-area trapezoid from the layer expression against the reference zoid list. It will return immediately on the first such zoid that is not fully covered by the reference zoid list.

The return value is 0 if there was only one trapezoid from the layer expression, and it did not overlap the reference zoid list. Otherwise, if all layer expression trapezoids were covered by the reference zoid list, 2 is returned, or 1 if not. Note that 1 will be returned if there is no intersection and more than one layer expression trapezoid. Use `TestCoveragePartial` to fully distinguish the not-full case. The present function is most efficient for determining when the layer expression dark area is or is not fully covered.

(int) `TestCoveragePartial(`*layer_expr*`, `*zoidlist*`, `*minsize*`)`
This function will return an integer value indicating the coverage of the layer expression given in the first argument over the regions described in the second argument. The first argument can be a string containing a layer expression, or a return from `ParseLayerExpression`. If the second argument is 0, the current reference zoidlist as set with `SetZref` is assumed. This defaults to tha area of the current cell.

The third argument is an integer which gives the minimum dimension in internal units of trapezoids which will be considered in the result. Sub-dimensional trapezoids are ignored. This minimizes false-positive tests due to "slivers" caused by clipping errors in non-Manhattan geometry. If the geomentry is known to be Manhattan, 0 can be used. If 45's only, 2 is recommended, otherwise 4. Negative values are taken as zero.

The function tests each dark-area trapezoid from the layer expression against the reference zoid list. It will return immediately on the first such zoid that is partially covered by the reference zoid list, of after finding both a fully covered zoid and a fully uncovered zoid.

The return value is 0 if there is no dark area from the layer expression that intersects the reference zoid list, 2 if the layer expression dark area falls entirely in the reference zoid list, and 1 if coverage is partial. This test is a bit expensive but provides definitive results,

(int) `TestCoverageNone(`*layer_expr*`,` *zoidlist*`,` *minsize*`)`

This function will return an integer value indicating the coverage of the layer expression given in the first argument over the regions described in the second argument. The first argument can be a string containing a layer expression, or a return from `ParseLayerExpression`. If the second argument is 0, the current reference zoidlist as set with `SetZref` is assumed. This defaults to tha area of the current cell.

The third argument is an integer which gives the minimum dimension in internal units of trapezoids which will be considered in the result. Sub-dimensional trapezoids are ignored. This minimizes false-positive tests due to "slivers" caused by clipping errors in non-Manhattan geometry. If the geomentry is known to be Manhattan, 0 can be used. If 45's only, 2 is recommended, otherwise 4. Negative values are taken as zero.

The function tests each dark-area trapezoid from the layer expression against the reference zoid list. It will return immediately on the first such zoid that is not completely uncovered by the reference zoid list.

The return value is 0 if there is no dark area from the layer expression that intersects the reference zoid list, 1 otherwise. This test is most efficient when determining whether or not the layer expression dark area intersects the reference list.

(int) `TestCoverage(`*layer_expr*`,` *zoidlist*`,` *testfull*`)`

This function is deprecated and should not be used in new scripts. The `TestCoverageFull`, `TestCoveragePartial`, and `TestCoverageNone` functions are replacements.

When the boolean *testfull* is true, this function is identical to `TestCoveragePartial` with a *minsize* value of 4. When *testfull* is false, this function is equivalent to `TestCoverageNone` again with a *minsize* of 4.

(object_handle) `ZtoObjects(`*zoidlist*`,` *lname*`,` *join*`,` *to_dbase*`)`

This function will create a list of objects from a zoidlist. The objects will be created on the layer whose name is given in the second argument, which will be created if it does not already exist. If this argument is 0, the current layer will be used. If the *join* argument is nonzero, the objects created will comprise a minimal set of polygons that enclose all of the trapezoids. If the *join* argument is 0, the objects will be have the same geometry as the individual trapezoids. If the *to_dbase* argument is nonzero, the new objects will be added to the database. Otherwise, the new objects will be "copies" that can be manipulated with other functions that accept object copies, but they will not appear in the database. The function will fail if not called in physical mode, or the layer could not be created.

(int) `ZtoTempLayer(`*longname*`,` *zoidlist*`,` *join*`)`

This function creates a temporary layer using *longname*, and adds the content of the *zoidlist* to the new layer, in the current cell. If the temporary layer for *longname* exists, it will be used, with existing geometry untouched. If *join* is nonzero, the zoidlist will be added as a minimal set of polygons, otherwise each zoid will be added as a box or polygon. The function returns 1 on success, 0 otherwise. This works in physical mode only.

(int) `ClearTempLayer(`*longname*`)`

This function will clear all of the objects in the current cell from the given layer, without saving them in the undo list. If successful, 1 is returned, otherwise 0 is returned. This works in physical mode only.

(int) `ZtoFile(`*filename*`,` *zoidlist*`,` *ascii*`)`

Save the zoidlist in a file, whose name is given in the first argument. The zoidlist can be recovered with `ZfromFile`.

There are two file formats available. If the boolean argument *ascii* is nonzero, a human-readable ASCII text file is produced. Each line contains the six numbers that describe a trapezoid, using the following C-style format string:

```
"yl=%d yu=%d ll=%d ul=%d lr=%d ur=%d"
```

The numbers are integer values in internal units (usually 1000 units per micron).

If the *ascii* argument is zero, the file is in OASIS format, using a single dummy cell (named “zoidlist”) and layer (“0100”), and uses only TRAPEZOID and CTRAPEZOID geometry records. The OASIS representation is more compact and is the appropriate choice for very large trapezoid collections.

The function returns 1 if successful, 0 otherwise.

(zoidlist) `ZfromFile(`*filename*`)`

Read the file, which was produced by `ZtoFile`, and return the list of trapezoids it contains. If an error occurs in reading or an interrupt is received, this function will fail (halting the script). Otherwise a zoidlist will always be returned, but the list may be empty.

(int) `ReadZfile(`*filename*`)`

This will read a trapezoid list file whose name is specified as the required string argument. This is an ASCII file consisting of two types of lines:

1. Trapezoid lines, in the ASCII format used by `ZfromFile` and produced by `ZtoFile`, i.e., in the format:

   ```
   yl=%d yu=%d ll=%d ul=%d lr=%d ur=%d
   ```

2. Layer designation lines in the form:

   `L` *layer_name*

   The *layer_name* should be an *Xic*-style name for a layer, the layer will be created if it does not exist.

When a layer designation line is encountered, the trapezoids that have been read since the file start or last layer designator are written into the current cell on the specified layer. Thus, each block of trapezoid lines must be followed by a layer designation line for the trapezoids to be recognized.

However, if the file contains no layer designation lines, all trapezoids will be added to the current cell on the current layer.

Lines that are not recognized as one of these two forms are ignored.

This function always returns 1. The function will fail if the file can not be opened.

(zoidlist) `ChdGetZlist(`*chd_name*`,` *cellname*`,` *scale*`,` *array*`,` *clip*`,` *all*`)`

This function will create and return a trapezoid list created from objects read through the Cell Hierarchy Digest (CHD) whose access name is given in the first argument.

See the table in 14.1 for the features that apply during a call to this function. An overall transformation can be set with `ChdSetFlatReadTransform`, in which case the area given applies in the “root” coordinates.

The *cellname*, if nonzero, must be the cell name after any aliasing that was in force when the CHD was created. If *cellname* is passed 0, the default cell for the CHD is understood. This is a cell name configured into the CHD, or the first top-level cell found in the archive file.

The *scale* factor will be applied to all coordinates. The accepted range is 0.001 – 1000.0.

If the *array* argument is passed 0, no windowing will be used. Otherwise the array should have four components which specify a rectangle, in microns, in the coordinates of *cellname*. The values are

array`[0]`	X left
array`[1]`	Y bottom
array`[2]`	X right
array`[3]`	Y top

If an array is given, only the objects and subcells needed to render the window will be processed.

If the boolean value *clip* is nonzero and an array is given, trapezoids will be clipped to the window. Otherwise no clipping is done.

If the boolean variable *all* is nonzero, the objects in the hierarchy under *cellname* will be transformed and added to the trapezoid list, i.e., the list will be a flat representation of the entire hierarchy. Otherwise, only objects in *cellname* are processed.

F.7.2 Operations

(zoidlist) `Filt(`*zoids*`,` *lexpr*`)`

This function is rather specialized. First, the trapezoids passed by the handle in the first argument are separated into groups of mutually-connected trapezoids. Each group is like a wire net. We throw out the groups that do not intersect with nonzero area the dark area implied by the layer expression second argument. The return value is a handle to a list of the trapezoids that remain.

(zoidlist) `GeomAnd(`*zoids1* `[,` *zoids2*`])`

This function takes either one or two arguments, each of which is taken as a zoidlist after possible conversion as described in the text for this section. If one argument is given, the return is a zoidlist consisting of the intersection regions between zoids in the argument list. If two arguments are given, the return is a list of intersecting regions between the two argument lists.

(zoidlist) `GeomAndNot(`*zoids1*`,` *zoids2*`)`

This function takes two arguments, each of which is taken as a zoidlist after possible conversion as described in the text for this section. The return is a list of regions covered by the first list that are not covered by the second.

(zoidlist) `GeomCat(`*zoids1* `[, ...])`

This function takes one or more arguments, each of which is taken as a zoidlist after possible conversion as described in the text for this section. The return is a list of all regions from each of the arguments. There is no attempt to clip or merge the returned list.

(zoidlist) `GeomNot(`*zoids*`)`

This function takes one argument, which is taken as a zoidlist after possible conversion as described in the text for this section. The return is a list of zoids representing the areas of the reference area not covered by the argument list.

(zoidlist) `GeomOr(`*zoids1*`, ...)`

This function takes one or more arguments, each of which is taken as a zoidlist after possible conversion as described in the text for this section. The return is a list of all regions from each of the arguments, merged and clipped so that no elements overlap.

(zoidlist) `GeomXor(`*zoids1* `[,` *zoids2*`])`

This function takes one or two arguments, each of which is taken as a zoidlist after possible conversion as described in the text for this section. If one argument is given, the return is a list of areas where one and only one zoid from the argument has coverage (note that this is not exclusive-or, in spite of the function name). If two arguments are given, the return is the exclusive-or of the two lists, i.e., the areas covered by either list but not both.

F.7.3 Spatial Parameter Tables

(int) `ReadSPtable(`*filename*`)`

This function reads a specification file for a spatial parameter table. A spatial parameter table is a two dimensional array of floating point values, which can be accessed via x-y coordinate pairs. The user can define any number of such tables, each of which is given a unique identifying keyword. Tables remain defined until explicitly destroyed, or until `ClearAll` is called.

The tables are input through a file, which uses the following format:

keyword X DX NX Y DY NY
X Y value
...

Blank lines and lines that begin with punctuation are ignored. There is one "header" line with the following entries:

keyword
Arbitrary word for identification. An existing database with the same identifier will be replaced.

X
Reference coordinate in microns.

DX
Grid spacing in X direction, in microns, must be > 0.

NX
Number of grid cells in X direction, must be > 0.

Y
Reference coordinate in microns.

DY
Grid spacing in Y direction, in microns, must be > 0.

NY
Number of grid cells in Y direction, must be > 0.

The header line is followed by data lines that supply a value to the cells. The X, Y given in microns specifies the cell. A second access to a cell will simply overwrite the data value for that cell. Unwritten cells will have a zero value.

The function returns 1 on success, 0 otherwise with an error message available from the `GetError` function.

(int) `NewSPtable(`*name*`,` *x0*`,` *dx*`,` *nx*`,` *y0*`,` *dy*`,` *ny*`)`

This will create a new, empty spatial parameter table in memory, replacing any existing table with the same name. The first argument is a string giving a short name for the table. The table origin is at *x0*, *y0* (in microns). The unit cell size is given by *dx*, *dy* in microns, and the number of cells along x and y is *nx*, *ny*.

The function returns 1 on success, 0 otherwise, with a message available from `GetError`.

(int) `WriteSPtable(`*name*`)`
This will write the named spatial parameter table to a file. The return value is 1 on success, 0 otherwise, with an error message available from `GetError`.

(int) `ClearSPtable(`*name*`)`
This will destroy the spatial parameter table whose keyword matches the string given. If a numeric 0 (`NULL`) or a null string is passed, all spatial parameter tables will be destroyed. The return value is the number of tables destroyed.

(int) `FindSPtable(`*name*`,` *array*`)`
This function returns 1 if a spatial parameter table with the given name exists in memory, 0 otherwise. The *array* is an array of size 6 or larger, or the constant 0. If an array name is passed, and the named table exists, the array is filled in with the following table parameters:

array`[0]`	origin x in microns
array`[1]`	x spacing in microns
array`[2]`	row size
array`[3]`	origin y in microns
array`[4]`	y spacing in microns
array`[5]`	column size

(real) `GetSPdata(`*name*`,` *x*`,` *y*`)`
This function returns the value from the spatial parameter table keyed by *name*, at coordinate *x*,*y* given in microns. If *x*,*y* is out of range, 0 is returned. The function fails (halts execution) if the table can't be found.

(int) `SetSPdata(`*name*`,` *x*`,` *y*`,` *value*`)`
This function will set the data cell corresponding to *x*,*y* (in microns) of the named spatial parameter table to the *value*. The return value is 1 if successful, 0 if *x*,*y* is out of range, or some other error occurs. The function fails (halts execution) if the table can't be found.

F.7.4 Polymorphic Flat Database

There functions are related to creating and using "special" databases. A special database is a spatially sorted container for objects or trapezoids (not cell instances or cells), with varying internal formats. The following script functions expose this functionality.

(int) `ChdOpenOdb(`*chd_name*`,` *scale*`,` *cellname*`,` *array*`,` *clip*`,` *dbname*`)`
This function will create a "special database" of the objects read through the Cell Hierarchy Digest (CHD) whose access name is passed as the first argument.

See the table in 14.1 for the features that apply during a call to this function. An overall transformation can be set with `ChdSetFlatReadTransform`, in which case the area given applies in the "root" coordinates.

The *scale* factor will be applied to all coordinates. The accepted range is 0.001 – 1000.0.

The *cellname*, if nonzero, must be the cell name after any aliasing that was in force when the CHD was created. If *cellname* is passed 0, the default cell for the CHD is understood. This is a cell name configured into the CHD, or the first top-level cell found in the archive file.

The *array*, if not 0, is an array of four values or larger giving a rectangular area of *cellname* to read. The values are in microns, in order L,B,R,T. If zero, the entire cell bounding box is understood. If the boolean value *clip* is nonzero, objects will be clipped to the array, if given. The *dbname* is a string which names the database. This can be any short name string. The database can be retrieved or cleared using this name.

The return value is 1 on success, 0 otherwise, with an error message likely available from `GetError`.

(int) `ChdOpenZdb(`*chd_name*`,` *scale*`,` *cellname*`,` *array*`,` *clip*`,` *dbname*`)`

This function will create a "special database" of the trapezoid representations of objects read through the Cell Hierarchy Digest (CHD) whose access name is passed as the first argument.

See the table in 14.1 for the features that apply during a call to this function. An overall transformation can be set with `ChdSetFlatReadTransform`, in which case the area given applies in the "root" coordinates.

The *scale* factor will be applied to all coordinates. The accepted range is 0.001 – 1000.0.

The *cellname*, if nonzero, must be the cell name after any aliasing that was in force when the CHD was created. If *cellname* is passed 0, the default cell for the CHD is understood. This is a cell name configured into the CHD, or the first top-level cell found in the archive file.

The *array*, if not 0, is an array of four values or larger giving a rectangular area of *cellname* to read. The values are in microns, in order L,B,R,T. If zero, the entire cell bounding box is understood. If the boolean value *clip* is nonzero, trapezoids will be clipped to the array, if given. The *dbname* is a string which names the database. This can be any short name string. The database can be retrieved or cleared using this name.

The return value is 1 on success, 0 otherwise, with an error message likely available from `GetError`.

(int) `ChdOpenZbdb(`*chd_name*`,` *scale*`,` *cellname*`,` *array*`,` *dbname*`,` *dx*`,` *dy*`,` *bx*`,` *by*`)`

This function will create a "special database" of the trapezoid representations of objects read through the Cell Hierarchy Digest (CHD) whose access name is passed as the first argument. This will open a database similar to `ChdOpenZdb`, however the trapezoids will be saved in binned lists.

See the table in 14.1 for the features that apply during a call to this function. An overall transformation can be set with `ChdSetFlatReadTransform`, in which case the area given applies in the "root" coordinates.

The *scale* factor will be applied to all coordinates. The accepted range is 0.001 – 1000.0.

The *cellname*, if nonzero, must be the cell name after any aliasing that was in force when the CHD was created. If *cellname* is passed 0, the default cell for the CHD is understood. This is a cell name configured into the CHD, or the first top-level cell found in the archive file.

The *array*, if not 0, is an array of four values or larger giving a rectangular area of *cellname* to read. The values are in microns, in order L,B,R,T. If zero, the entire cell bounding box is understood. The *dbname* is a string which names the database. This can be any short name string. The database can be retrieved or cleared using this name.

The *dx*, *dy* are the grid spacing values for the bins, in microns. These values must be positive. The *bx*, *by* are non-negative overlap bloat values for the bins. The actual bins are bloated by these values in the x and y directions. The trapezoids will be clipped to the bins.

The return value is 1 on success, 0 otherwise, with an error message likely available from `GetError`.

(object_handle) `GetObjectsOdb(`*dbname*`,` *layer_list*`,` *array*`)`

This returns a handle to a list of objects, extracted from a named database created with `ChdOpenOdb`. The first argument is a database name string as given to `ChdOpenOdb`. This function will work only with databases produced by that function.

The second argument is a string containing a space-separated list of layer names, or 0. Objects for each of the given layers will be obtained. Objects on the same layer will be grouped together, with groups ordered as in the *layer_list*. If this argument is 0, all layers will be used, ordered bottom-up as in the layer table.

The third argument is an array, as passed to `ChdOpenOdb`, or 0. If 0, all objects for the specified layers in the database will be retrieved. Otherwise, only those objects with bounding boxes that overlap the array rectangle with nonzero area will be retrieved. The objects retrieved are copies of the database objects, which are not affected.

(stringlist_handle) `ListLayersDb(`*dbname*`)`
This function returns a handle to a list of layer name strings, naming the layers used in the database. It applies to all of the database types. On error, a scalar 0 is returned.

(zoidlist) `GetZlistDb(`*dbname*`, `*layer_name*`, `*zoidlist*`)`
This returns a zoidlist associated with a layer, extracted from a named database created with `ChdOpenOdb`, `ChdOpenZdb`, or `ChdOpenZbdb`. The first argument is a database name string as given to `ChdOpenOdb` or equivalent. The second argument is the associated layer name.

The third argument is the reference trapezoid list. If the database was opened with `ChdOpenOdb` or `ChdOpenZdb`, the returned zoidlist will be clipped to the reference list. If the database was opened with `ChdOpenZbdb`, the trapezoids for the bin containing the center of the first trapezoid in the reference list will be returned. In all cases, the returned trapezoids are copies, the database is not affected.

See also the `GetZlist` function, which can work similarly.

(zoidlist) `GetZlistZbdb(`*dbname*`, `*layer_name*`, `*nx*`, `*ny*`)`
Return the zoidlist for the given bin and layer. This applies only to databases opened with `ChdOpenZbdb`. The 0,0 bin is in the lower left corner.

(int) `DestroyDb(`*dbname*`)`
This function will free and clear the special database named in the argument. This is the database name as given to `ChdOpenOdb` or equivalent. If the argument is 0, then all special databases will be freed and cleared. This function always returns 1.

(int) `ShowDb(`*dbname*`, `*array*`)`
This function will pop up a window displaying the area given in the array of the special database named in *dbname*. The array argument is in the same format as passed to `ChdOpenOdb` or equivalent. If passed 0, the bounding box containing all objects in the database is understood. The return value is the window number of the new window (1–4) or -1 if an error occurred.

F.7.5 Named String Tables

This interface provides general purpose string hash tables. The hash tables are useful for saving and retrieving a string-keyed integer value, and for detecting or preventing the occurrence of duplicate strings in a list. The hash tables are persistent until explicitly freed, i.e., they remain in memory after a script completes (if not destroyed), and can be invoked by subsequent scripts. Each hash table is accessed by an arbitrary user-supplied name, and there is no limit on the number of tables that can be created.

(int) `FindNameTable(`*tabname*`, `*create*`)`
This function will create or verify the existence of a named string hash table. The named tables are available for use in scripts, for associating a string with an integer and for efficiently ensuring uniqueness in a collection of strings. The named tables persist until explicitly destroyed.

The *tabname* is an arbitrary name token used to access a named hash table. This function returns 1 if the named hash table exists, 0 otherwise. If the boolean argument *create* is nonzero, if the named table does not exist, it will be created, and 1 returned.

(int) `RemoveNameTable(`*tabname*`)`
This function will destroy a named hash table, as created with `FindNameTable` in create mode. It the table exists, it will be destroyed, and 1 is returned. If the given name does not match an existing table, 0 is returned.

(stringlist_handle) `ListNameTables()`
This function returns a handle to a list of names of named hash tables currently in memory.

(int) `ClearNameTables()`
This functions destroys all named hash tables in memory.

(int) `AddNameToTable(`*tabname*`, `*name*`, `*value*`)`
This will add a string and associated integer to a named hash table. The hash table whose name is given as the first argument must exist in memory, as created with `FindNameTable` in create mode. The *name* can be any non-null and non-empty string. The *value* can be any integer, however, the value -1 is reserved for internal use as a “not in table” indication.

If *name* is inserted into the table, 1 is returned. If *name* already exists in the table, or the table does not exist, 0 is returned. The *value* is ignored if the *name* already exists in the table, the existing value is not updated.

(int) `RemoveNameFromTable(`*tabname*`, `*name*`)`
This will remove the *name* string from the named hash table whose name is given as the first argument. If the *name* string is found and removed, 1 is returned. Otherwise, 0 is returned.

(int) `FindNameInTable(`*tabname*`, `*name*`)`
This function will return the data value saved with the *name* string in the table whose name is given as the first argument. If the table is not found, or the *name* string is not found, -1 is returned. Otherwise the returned value is that supplied to `AddNameToTable` for the *name* string. Note that it is a bad idea to use -1 as a data value.

(stringlist_handle) `ListNamesInTable(`*tabname*`)`
This function returns a handle to a list of the strings saved in the hash table whose name is supplied as the first argument.

F.8 Design Rule Checking Functions

F.8.1 DRC

The following functions relate to the design rule checking subsystem.

(int) `DRCstate(`*state*`)`
This function sets the interactive DRC state, and returns the existing state. If the argument is 0, interactive DRC is turned off. If nonzero, interactive DRC is turned on. If greater than 1, error messages will not pop up. The return value is the present state, which is a value of 0–2, similarly interpreted.

(int) `DRCsetLimits(`*batch_cnt*`, `*intr_cnt*`, `*intr_time*`, `*skip_cells*`)`
Deprecated in favor of DRCsetMaxErrors and similar.

This function sets the limits used in design rule checking. Each argument, if negative, will cause the related value to be unchanged by the function call. For the first three arguments, the value "0" is interpreted as "no limit".

batch_cnt
This sets the maximum number of errors to record in batch-mode error checking. When this number is reached, the checking is aborted. Values 0 – 100000 are accepted.

intr_cnt
This sets the maximum number of objects tested in interactive DRC. The testing aborts when this count is reached. Values of 0 – 100000 are accepted.

intr_time
This sets the maximum time allowed for interactive DRC testing. The value given is in milliseconds, and values of 0 – 30000 are accepted.

skip_cells
If nonzero, testing of newly placed, moved, or copied subcells is skipped in interactive DRC. If zero, subcells will be tested. This can be a lengthly operation.

This function always returns 1. Out-of-range arguments are set to the maximum permissible values.

(int) `DRCgetLimits(`*array*`)`
Deprecated in favor of DRCgetMaxErrors and similar.

This function fills the *array*, which must have size 4 or larger, with the current DRC limit values. These are, in order,

[0] The batch error count limit.
[1] The interactive object count limit.
[2] The interactive time limit in milliseconds.
[3] A flag which indicates interactive DRC is skipped for subcells.

The return value is always 1. The function fails if the array argument is bad.

(int) `DRCsetMaxErrors(`*value*`)`
Set the maximum violation count allowed before a batch DRC run is terminated. If set to 0, no limit is imposed. The value is clipped to the acceptable range 0 – 100,000. If not set, a value 0 (no limit) is assumed. The function returns the previous value.

(int) `DRCgetMaxErrors()`
Returns the maximum violation count before a batch DRC run is terminated. If set to 0, no limit is imposed.

(int) `DRCsetInterMaxObjs(`*value*`)`
Set the maximum number of objects tested in interctive DRC. Further testing is skipped when this value is reached. A value of 0 imposes no limit. The passed value is clipped to the acceptable range 0 – 100,000, the value used if not set is 1000. The function returns the previous setting.

(int) `DRCgetInterMaxObjs()`
Return the maximum number of objects tested in interctive DRC. Further testing is skipped when this value is reached. A value of 0 imposes no limit.

(int) `DRCsetInterMaxTime(`*value*`)`
Set the maximum time in milliseconds allowed for interactive DRC testing after an operation. The

testing will abort after this limit, returning program control to the user. If set to 0, no time limit is imposed. the passed value is clipped to the acceptable range 0 - 30,000. If not set, a value of 5000 (5 seconds) is used. The function returns the previous value.

(int) `DRCgetInterMaxTime()`
Return the maximum time in milliseconds allowed for interactive DRC testing after an operation. The testing will abort after this limit, returning program control to the user. If set to 0, no time limit is imposed.

(int) `DRCsetInterMaxErrors(`*value*`)`
Set the maximum number of errors allowed in interactive DRC testing after an operation. Further testing is skipped after this count is reached. A value of 0 imposes no limit. The value will be clipped to the acceptable rnge 0 – 1000. If not set, a value of 100 is used. The function returns the previous value.

(int) `DRCgetInterMaxErrors()`
Return the maximum number of errors allowed in interactive DRC testing after an operation. Further testing is skipped after this count is reached. A value of 0 imposes no limit.

(int) `DRCsetInterSkipInst(`*value*`)`
If the boolean argument is nonzero, cell instances will not be checked for violations in interactive DRC. The test can be lengthly and the user may want to defer such testing. The return value is 0 or 1 representing the previous setting.

(int) `DRCgetInterSkipInst()`
The return value of this function is 0 or 1 representing whether cell instances are skipped (if 1) in interactive DRC testing.

(int) `DRCsetLevel(`*level*`)`
This function sets the DRC error recording level to the argument. The argument is interpreted as follows:

0 or negative	One error is reported per object.
1	One error of each type is reported per object.
2 or larger	All errors are reported.

This function always succeeds, and the previous level (0, 1, 2) is returned.

(int) `DRCgetLevel()`
This function returns the current error reporting level for design rule checking. Possible values are

0	One error is reported per object.
1	One error of each type is reported per object.
2	All errors are reported.

This function always succeeds.

(int) `DRCcheckArea(`*array*`,` *file_handle_or_name*`)`
This function performs batch-mode design rule checking in the current cell.

The *array* argument is an array of size 4 or larger, or 0 can be passed for this argument. If an array is passed, it represents a rectangular area where checking is performed, and the values are in microns in order L,B,R,T. If 0 is passed, the entire area of the current cell is checked.

The second argument can be a file handle opened with the `Open` function for writing, or the name of a file to open, or an empty string, or a null string or (equivalently) the scalar 0. This sets the destination for error recording. If the argument is null or 0, a file will be created in the current directory using the name template "`drcerror.log.`*cellname*", where *cellname* is the current cell.

If an empty string is passed (give `""` as the argument), output will go to the error log, and appear in the pop-up which appears on-screen. If a string is given, it is taken as a file name to open.

The function returns an integer, either the number of errors found or -1 on error. If -1 is returned, an error message is probably available from the `GetError` function.

(int) `DRCchdCheckArea(`*chdname*`,` *cellname*`,` *gridsize*`,` *array*`,` *file_handle_or_name*`,` *flatten*`)`

This function performs a batch-mode DRC of the given top-level cell, from the Cell Hierarchy Digest (CHD) whose access name is given as the first argument. Unlike other DRC commands, this function does not require that the entire layout be in memory, thus it is theoretically possible to perform DRC on designs that are too large for available memory.

If the given *cellname* is null or 0 is passed, the default cell for the named CHD is assumed.

The checking is performed on the areas of a grid, and only the cells needed to render the grid area are read into memory temporarily. The gridsize argument gives the size of this grid, in microns. If 0 is passed, no grid is used, and the entire layout will be read into memory, as in the normal case. If a negative value is passed, the value associated with the DrcPartitionSize variable is used. The chosen grid size should be small enough to avoid page swapping, but too-small of a grid will lengthen checking time (larger is better in this regard). The user can experiment to find a reasonable value for their designs. A good starting value might be 400.0 microns.

The *array* argument is an array of size 4 or larger, or 0 can be passed for this argument. If an array is passed, it represents a rectangular area where checking is performed, and the values are in microns in order L,B,R,T. If 0 is passed, the entire area of the *cellname* is checked.

The *file_handle_or_name* argument can be a file handle opened with the `Open` function for writing, or the name of a file to open, or an empty or null string or the scalar 0. This sets the destination for error recording. If the argument is null, empty or 0, a file will be created in the current directory using the name template "`drcerror.log.`*cellname*", where *cellname* is the top-level cell being checked. If a string is given, it is taken as a file name to open. There is no provision for sending output to the on-screen error logger, unlike in the `DRCcheckArea` function.

If the boolean argument *flatten* is true, the geometry will be flattened as it is read into memory. This will make life simpler and faster for the DRC evaluation functions, at the expense of (probably) much larger memory use. The user can experiment to find if this option provides any speed benefit.

The function returns an integer, either the number of errors found or -1 on error. If -1 is returned, an error message is probably available from the `GetError` function.

(int) `DRCcheckObjects(`*file_handle*`)`

This function checks each selected object for design rule violations. The *file_handle* argument is a file descriptor returned from the `Open` function, or 0. If a file descriptor is passed, output goes to that file, otherwise output goes to the on-screen error logger. This function returns the number of errors found.

(expr_handle) `DRCregisterExpr(`*expr*`)`

This function creates and tags a parse tree from the string argument, which is a layer expression, for later use, and returns a handle to the expression. This avoids the overhead of parsing the expression on each function call. The returned value is used by other functions (currently just the two below).

(int) `DRCtestBox(`*left*`,` *bottom*`,` *right*`,` *top*`,` *expr_handle*`)`

This function tests a rectangular area specified by the first four arguments for regions where a layer expression is true. The *expr_handle* argument is the handle of a layer expression returned by `DRCregisterExpr`. The returned value is 0 if the expression is nowhere true, 1 if the expression is true somewhere but not everywhere, and 2 if the expression is true everywhere in the test region.

(int) `DRCtestPoly(`*num*`,` *points*`,` *expr_handle*`)`

This function tests a polygon area for regions where a layer expression is true. The first argument is the number of points in the polygon. The second argument is the name of an array variable containing the polygon data. The polygon data are stored sequentially as x,y pairs, and the last point must be the same coordinate as the first. The length of the vector must be at least two times the value passed for the first argument. The *expr_handle* argument is the handle of a layer expression returned by `DRCregisterExpr`. The returned value is 0 if the expression is nowhere true, 1 if the expression is true somewhere but not everywhere, and 2 if the expression is true everywhere in the test region.

(zoidlist) `DRCzList(`*layername*`,` *rulename*`,` *index*`,` *source*`)`

This function will access existing design rule definitions, and use the associated test region generator to create a new trapezoid list, which is returned. For example, in a `MinSpaceTo` rule test, we construct a "halo" around source polygons. If this halo intersects any target polygons, a violation would be flagged. The list of trapezoids that constitute the halos around the source polygons is the return of this function.

The first three arguments specify an existing design rule. The rule is defined on the layer named in the first argument (a string). The type of rule is given as a string in the second argument. This is the name of an "edge" rule, which uses test regions constructed along edges to evaluate the rule. Valid names are the user-defined rules and

```
MinEdgeLength
MaxWidth
MinWidth
MinSpace
MinSpaceTo
MinSpaceFrom
MinOverlap
MinNoOverlap
```

The third argument is an integer index which specifies the rule to choose if there is more than one of the named type assigned to the layer. The index is zero based, and indicates the position of the rule when listed in the window of the **Design Rule Editor** panel from the **Edit Rules** button in the **DRC Menu**, relative to and counting only rules of the same type. The is also the order as first seen by *Xic*, as read from the technology file or created interactively.

The fourth argument is a "zoidlist" as is taken by many of the functions that deal with layer expressions and trapezoid lists, as explained for those functions (see F.7.1). If the value passed is a scalar 0, then geometry is obtained from the full hierarchy of the current cell. In this case, the created test areas will be identical to those created during a DRC run. It may be instructive to create a visible layer from this result, to see where testing is being performed.

If the argument instead passes trapezoids, the result will be creation of the test regions as if the passed trapezoids were features on the layer or `Region` associated with the rule. The actual features on the layer are ignored.

The function will fail and halt execution if the first three arguments do not indicate an existing design rule definition.

(zoidlist) `DRCzListEx(`*source*`,` *target*`,` *inside*`,` *outside*`,` *incode*`,` *outcode*`,` *dimen*`)`

This is similar to `DRCzList`, however it does not reference an existing rule. Instead, it accesses the test area generator directly, effectively creating an internal, temporary rule.

The first argument is a "zoidlist" as expected by other functions that accept this argument type (see F.7.1). Unlike for `DRCzList`, this argument can not be zero or null.

The second argument is a string providing a target layer expression. This may be scalar 0 or null. The *inside* and *outside* arguments are strings providing layer expressions that will select which parts of an edge will be used for test area generation. The *inside* is the area inside the figure at the edge, and *outside* is just outside of the figure along the edge. Either can be null or scalar 0.

The *incode* and *outcode* are integer values 0–2 which indicate how the inside and outside expressions are to be interpreted with regard to defining the "active" part of the edge. The values have the following interpretations:

0	Don't care, the value expression is ignored.
1	The active parts of the edge are where the expression is clear.
2	The active parts of the edge are where the expression is dark.

The *dimen* is the width of the test area, in microns. It must be a positive real number.

If all goes well, a trapezoid list reprseenting the effective test areas is returned.

F.9 Extraction Functions

F.9.1 Menu Commands

The functions in this section provide an interface to the extraction system. This interface is by no means complete, but it allows many common operations to be performed and allows traversal and information retrieval.

(int) `DumpPhysNetlist(`*filename*`,` *depth*`,` *modestring*`,` *names*`)`

This function dumps a netlist file extracted from the physical part of the database, much like the **Dump Phys Netlist** command in the **Extract Menu**. The *filename* argument is a file name which will receive the output. If null or empty, the file will be the base name of the current cell with "`.physnet`" appended. The *depth* argument specifies the depth of the hierarchy to process. If an integer, 0 represents the current cell only, 1 includes the first level subcells, etc. A negative integer specifies to process the entire hierarchy. This argument can also be a string beginning with the letter '`a`', which will process all levels of the hierarchy.

The third argument is a string, consisting of characters from the table below, which set the mode of the command. These are analogous to the check boxes that appear with the **Dump Phys Netlist** command. If a character does not appear in the string, that option is turned off. If it appears in lower case, the option is turned on, and if it appears in upper case, the option will be set by the present value of the corresponding **!set** variable. The characters can appear in any order.

character	option	corresponding variable
`n`	**net**	PnetNet
`d`	**devs**	PnetDevs
`s`	**spice**	PnetSpice
`b`	**list bottom-up**	PnetBottomUp
`g`	**show geometry**	PnetShowGeometry
`c`	**include wire cap**	PnetIncludeWireCap
`a`	**list all cells**	PnetListAll
`l`	**ignore labels**	PnetNoLabels

The final argument, if not null or empty, contains a space-separated list of physical format names, each of which must match a `PnetFormat` name in the format library file, or option names from the table above. The names that contain white space should be double-quoted.

For each cell, a field in the output is generated for each format choice implicit in the *modestring* or given in the *names*. In most cases, only one format is probably wanted. The option text in the table above can also be included in the *names*, which is equivalent to giving the corresponding lower-case letter in the *modestring*. The *modestring* setting will have precedence if there is a conflict. If both the *modestring* and the *names* string are empty or null, an effective mode string consisting of all of the upper-case option letters is used.

Example: print a SPICE file

```
DumpPhysNetlist("myfile.cir", "a", "s", 0)
or
DumpPhysNetlist("myfile.cir", "a", 0, "spice")
```

If the function succeeds, 1 is returned, otherwise 0 is returned.

(int) `DumpElecNetlist(`*filename*`, `*depth*`, `*modestring*`, `*names*`)`

This function dumps a netlist file extracted from the electrical part of the database, much like the **Dump Elec Netlist** command in the **Extract Menu**. The *filename* argument is a file name which will receive the output. If null or empty, the file will be the base name of the current cell with "`.elecnet`" appended. The *depth* argument specifies the depth of the hierarchy to process. If an integer, 0 represents the current cell only, 1 includes the first level subcells, etc. A negative integer specifies to process the entire hierarchy. This argument can also be a string beginning with the letter '`a`', which will process all levels of the hierarchy.

The third argument is a string, consisting of characters from the table below, which set the mode of the command. These are analogous to the check boxes that appear with the **Dump Elec Netlist** command. If a character does not appear in the string, that option is turned off. If it appears in lower case, the option is turned on, and if it appears in upper case, the option will be set by the present value of the corresponding **!set** variable. The characters can appear in any order.

character	option	corresponding variable
`n`	**net**	EnetNet
`s`	**spice**	EnetSpice
`b`	**list bottom-up**	EnetBottomUp

The final argument, if not null or empty, contains a space-separated list of electrical format names, each of which must match an `EnetFormat` name in the format library file, or option names from the table above. The names that contain white space should be double quoted.

For each cell, a field in the output is generated for each format choice implicit in the *modestring* or given in the *names*. In most cases, only one format is probably wanted. The option text in the table above can also be included in the *names*, which is equivalent to giving the corresponding lower-case letter in the *modestring*. The *modestring* setting will have precedence if there is a conflict. If both the *modestring* and the *names* string are empty or null, an effective mode string consisting of all of the upper-case option letters is used.

If the function succeeds, 1 is returned, otherwise 0 is returned.

(int) `SourceSpice(`*filename*`, `*modestring*`)`

This function will parse a SPICE file, adding to or updating the electrical part of the database with the devices and subcircuits found. This is equivalent to the **Source SPICE** command in the **Extract Menu**. The first argument is a path to the SPICE file to process.

The final argument is a string, consisting of characters from the table below, which set the mode of the command. These are analogous to the check boxes that appear with the **Source SPICE** command. If a character does not appear in the string, that option is turned off. If it appears in lower case, the option is turned on, and if it appears in upper case, the option will be set by the present value of the corresponding **!set** variable. The characters can appear in any order. If the string is empty or null, all options will be set by the corresponding variables.

character	option	corresponding variable
`a`	**all devs**	SourceAllDevs
`r`	**create**	SourceCreate
`l`	**clear**	SourceClear

If the operation succeeds, 1 is returned, otherwise 0 is returned.

(int) `ExtractAndSet(`*depth*`,` *modestring*`)`

This function performs extraction on the physical part of the database, updating the electrical part. This is equivalent to the **Source Physical** command in the **Extract Menu**. The first argument indicates the depth of the hierarchy to process. This can be an integer: 0 means process the current cell only, 1 means process the current cell plus the subcells, etc., and a negative integer sets the depth to process the entire hierarchy. This argument can also be a string starting with '`a`' such as "`a`" or "`all`" which indicates to process the entire hierarchy.

The final argument is a string, consisting of characters from the table below, which set the mode of the command. These are analogous to the check boxes that appear with the **Source Physical** command. If a character does not appear in the string, that option is turned off. If it appears in lower case, the option is turned on, and if it appears in upper case, the option will be set by the present value of the corresponding **!set** variable. The characters can appear in any order. If the string is empty or null, all options will be set by the corresponding variables.

character	option	corresponding variable
`a`	**all devs**	NoExsetAllDevs
`r`	**create**	NoExsetCreate
`l`	**clear**	ExsetClear
`c`	**include wire cap**	ExsetIncludeWireCap
`n`	**ignore labels**	ExsetNoLabels

If the operation succeeds, 1 is returned, otherwise 0 is returned. This function does not redraw the windows.

(object_handle) `FindPath(`*x*`,` *y*`,` *depth*`,` *use_extract*`)`

This function returns a handle to a list of copies of physical conducting objects in a wire net. The *x*,*y* point (microns, in the physical part of the current cell) should intersect a conducting object, and the list will consist of this object plus connected objects. The *depth* argument is an integer or a string beginning with "`a`" (for "all") which gives the hierarchy search depth. Only objects in cells to this depth will be considered for addition to the list (0 means objects in the current cell only). If the boolean value *use_extract* is nonzero, the main extraction functions will be used to determine the connectivity. If the value is zero, the connectivity is established through geometry. This is similar to the **Select Path** and **"Quick" Path** modes available in the **Path Selection Control** panel.

The return value is a handle to a list of object copies, or 0 if no objects are found.

(object_handle) `FindPathOfGroup(`*groupnum*`,` *depth*`)`

This function returns a handle to a list of copies of physical conducting objects in the group number from the current cell given, to the given depth. The depth argument is an integer or a string beginning with "`a`" (for "`all`") which gives the hierarchy search depth. Only objects in cells to this depth will be considered for addition to the list (0 means objects in the current cell only).

The function will fail (halt the script) on a major error. If the group number is out of range, or a "minor" error occurs, the function will return a scalar 0, and an error message should be available from `GetError`.

Otherwise, the return value is a handle to a list of object copies, or the list may be empty if the group has no physical objects.

F.9.2 Terminals

Here, a "terminal" refers to a `node` property of an electrical device or circuit. Both masters and instances have such properties, though their internal structure differs a bit. A "terminal_handle" is a handle to a list of terminals, that can be passed to functions that provide information about or operate on node properties.

In the next section, we introduce "physical terminals", which are different objects. A physical terminal is a data structure that stores information about the physical aspects of a terminal, including its location in the layout, an object that it may be bound to, and the associated layer. If a schematic has a layout and has been associated, then each terminal (node property) has a pointer to the corresponding physical terminal, and vice-versa. Thus, in general either object can be used to reference data. In fact, in most of the functions in this section and the next, the "handle" argument can be a handle to either a node property or physical terminal.

However, cells that are electrical-only will not have physical terminals, and similarly, a layout without a corresponding schematic will lack node properties. In these cases, only the existing object type can be used.

(terminal_handle) `ListTerminals()`
Return a handle containing a list of the connection terminals of the current cell. These correspond to the normal contact terminals as would be defined with the **subct** command, as represented by `node` properties of the electrical cell view. On success, a handle is returned containing the terminal list. If there are no terminals defined or some other error occurs, a scalar 0 is returned.

(terminal_handle) `FindTerminal(`*name*`,` *index*`,` *use_e*`,` *xe*`,` *ye*`,` *use_p*`,` *xp*`,` *yp*`)`
This function will return a handle referencing a single terminal, if one can be found among the current cell contact terminals that matches the arguments. The arguments specify parameters, any of which can be ignored. The non-ignored parameters must all match.

The *name* can be a string that will match an applied terminal name (not a default name generated by *Xic*). The argument will be ignored if a scalar 0 or null or empty string is passed.

The *index* is the terminal order index, or -1 if the parameter is to be ignored. This is the number that is shown within the terminal box in the **subct** command.

If *use_e* is a nonzero value, the next two arguments are taken as a location in the electrical drawing. These are specified in fictitious "microns" which represent 1000 internal units. These are the numbers displayed in the coordinate readout area while a schematic is being edited. A location match will depend of whether the electrical cell is symbolic or not. If symbolic, a location match to any of the placement locations will count as a match (terminals can have more than one "hot spot" in the symbolic display). If *use_e* is 0, the two arguments that follow are ignored and can be any numeric values.

Similarly, if *use_p* is nonzero, the next two arguments represent a coordinate in the layout, given in (real) microns. If a physical terminal is placed at the given location, a match will be indicated. If *use_p* is zero, the two arguments that follow are ignored, and can be set to any numeric values.

The arguments should provide at least one matchable parameter. Internally, the list of terminals is scanned, and the first matching terminal found is returned, referenced by a handle. If no terminals match, a scalar zero is returned.

(terminal_handle) `CreateTerminal(`*name*`,` *x*`,` *y*`,` *termtype*`)`

This function will create a new terminal in the schematic of the current cell. If a *name* string is passed, the terminal will be given that name. If this argument is a scalar 0 or a null or empty string, the terminal will not have an assigned name but will use an internally generated name. The terminal will be placed at the location indicated by the x and y arguments, which are in fictitious "microns" representing 1000 database units. These are the same coordinates as displayed in the coordinate readout while a schematic is being edited.

The *termtype* argument can be a scalar integer or a keyword, from the list below. This will assign a type to the terminal. The type is not used by *Xic*, but this facility may be useful to the user.

0	`input`
1	`output`
2	`inout`
3	`tristate`
4	`clock`
5	`outclock`
6	`supply`
7	`outsupply`
8	`ground`

Keyword matching is case-insensitive. If the argument is not recognized, and the default "`input`" will be used.

The function returns a handle that references the new terminal on success, or a scalar zero otherwise.

(int) `DestroyTerminal(`*thandle*`)`

This function will destroy the terminal referenced by the passed handle, and will close the handle. This destroys the terminal, which is actually a `node` property of the electrical current cell, and the linkage into the physical layout, if any. If a terminal was destroyed, value one is returned, or zero on error.

(string) `GetTerminalName(`*thandle*`)`

Return a string containing the name of the terminal or physical terminal referenced by the handle passed as an argument. Both objects have name fields that track. However, if no name was assigned, for a terminal a default name generated by *Xic* is returned, whereas the return from a physical terminal will be null.

(int) `SetTerminalName(`*thandle*`,` *name*`)`

The first argument is a handle that references a terminal or physical terminal. The second argument is a string which gives a name to apply. It can also be a scalar 0, or if null or empty any existing assigned name will be removed. Both terminals and physical terminals have names that track, this will change both, when both objects exist. The return value is one on success, zero if error.

(int) `GetTerminalType(`*thandle*`)`

Return a type code for the terminal referenced by the handle passed as an argument, which can also be a handle to the corresponding physical terminal. A non-negative return represents success. The code represents the terminal type set by the user. The terminal type is not used by *Xic*, but is available for user applications. The defined types are listed below. The default is type 0.

0	`input`
1	`output`
2	`inout`
3	`tristate`
4	`clock`
5	`outclock`
6	`supply`
7	`outsupply`
8	`ground`

(int) `SetTerminalType(`*thandle*`,` *termtype*`)`

This function will apply a terminal type to the terminal referenced by the handle passed as the first argument, which can also be a handle to the corresponding physical terminal. The second argument is either an integer, or a string keyword, from the list below.

0	`input`
1	`output`
2	`inout`
3	`tristate`
4	`clock`
5	`outclock`
6	`supply`
7	`outsupply`
8	`ground`

The function returns one if the type is set successfully, zero otherwise.

(int) `GetTerminalFlags(`*thandle*`)`

Return the flags for the terminal referenced by the handle passed as an argument, which can also be a handle to the corresponding physical terminal. The return value is an integer with bits representing flags as listed in the table below. On error, the return value is -1.

`-x1` (`BYNAME`)
: The terminal makes connections in the schematic by name rather than by location.

`0x2` (`VIRTUAL`)
: No longer used, reserved.

`0x4` (`FIXED`)
: The physical terminal has been placed by the user, and *Xic* should never move it.

`0x8` (`SCINVIS`)
: The electrical terminal will not be shown in schematics.

`0x10` (`SYINVIS`)
: The electrical terminal will not be shown in the symbol.

`0x100` (`UNINIT`)
: The terminal is not initialized (internal).

`0x200` (`LOCSET`)
: The physical terminal location has not been set (internal).

`0x400` (`POINTS`)
: Set when the terminal has multiple hot-spots.

`0x800` (`NOPHYS`)
: Set if the terminal has no physical implementation, such as a temperature node. Such terminals have no physical terminals.

(int) `SetTerminalFlags(`*thandle*`,` *flags*`)`
This will set the first five flags listed for `GetTerminalFlags` in the terminal referenced by the first argument, which can also be a handle to the corresponding physical terminal. All but the five least significant bits in the *flags* integer are ignored. The bits that are set will set the corresponding flag in the terminal, unset bits are ignored. The value one is returned on success, zero otherwise.

(int) `UnsetTerminalFlags(`*thandle*`,` *flags*`)`
This will unset the first five flags listed for `GetTerminalFlags` in the terminal referenced by the first argument, which can also be a handle to the corresponding physical terminal. All but the five least significant bits in the *flags* integer are ignored. The bits that are set will unset the corresponding flag in the terminal, unset bits are ignored. The value one is returned on success, zero otherwise.

(int) `GetElecTerminalLoc(`*thandle*`,` *index*`,` *array*`)`
This will return terminal locations in the electrical schematic, of the terminal referenced by the first argument. This argument can also be a handle to the corresponding physical terminal. The return is dependent on whether the electrical cell is symbolic or not. Values for x and y are returned in the *array*, which must have size two or larger. The returned values are in fictitions "microns" that correspond to 1000 database units. This is the same coordinate system indicated by the coordinate readout when editing a schematic.

If the electrical cell is not symbolic, the integer *index* argument must be zero, and the terminal location in the schematic is returned.

If the electrical cell is symbolic, there can be arbitrarily many "copies" of the terminal, representing multiple "hot spots" where the terminal can make connections. The *index* argument is a 0-based index for these locations. To get all of the locations, one should call this function repeatedly while incrementing the index from zero. A return value of zero indicates that the index is out of range (or some error occurred). A return value of one indicates success, with the array containing the location.

(int) `SetElecTerminalLoc(`*thandle*`,` *x*`,` *y*`)`
This function specifies a location for the terminal referenced by the first argument, for use in electrical mode. The x and y are coordinates in fictitions "microns" which are 1000 database units. This is the same coordinate system used in the coordinate readout when editing a schematic. As for most of these functions, the first argument can also be a handle to the corresponding physical terminal.

The function behaves differently depending on whether the electrical current cell is symbolic or not. If the electrical current cell not symbolic, the passed coordinates set the terminal location within the schematic. Otherwise, in symbolic mode, there can be arbitrarily many locations set. The function will add the passed location to the list of locations for the terminal, if it is not already using the location.

The function returns one on success, zero otherwise.

(int) `ClearElecTerminalLoc(`*thandle*`,` *x*`,` *y*`)`
This function applies only when the electric current cell is in symbolic mode. When true, a terminal may be displayed in arbitrarily many locations, representing different possible connection points. The x and y are coordinates in fictitions "microns" which are 1000 database units. This is the same coordinate system used in the coordinate readout when editing a schematic. If the coordinates match a hot spot of the terminal, that location is deleted.

It is not possible to delete the last location, there is always at least one active location. Calling this function when the electrical current cell is not symbolic has no effect. The function returns one on success, zero if error.

As for most of these functions, the first argument can also be a handle to the corresponding physical terminal.

F.9.3 Physical Terminals

As noted in the description in the previous section, physical terminals are a separate data structure that save layout information about the terminal, such as effective location, the layer attached to, or an object attached to. When a schematic exists and has been associated, the physical terminal and the electrical node property are linked, so access to one automatically provides access to the other. Thus, most of the the functions in this section that access physical terminals will also take a handle to a regular terminal equivalently, as did the functions in the previous section. However, if one data type does not exist, for the function to succeed, the exissting data type must be passed, and it must contain the data to be accessed.

Physical terminals that correspond to cell connection points are stored with the physical data, and are therefor potentially available when there is no schematic. Most commonly, however, they are created upon reading the electrical data for a cell.

(physterm_handle) `ListPhysTerminals()`
This returns a handle to a list of physical terminal structures that correspond to the cell connection points, as obtained from the physical part of the current cell.

(physterm_handle) `FindPhysTerminal(`*name*`,` *use_p*`,` *xp*`,` *yp*`)`
This attempts to find a physical terminal structure by name or location. If a name is given, i.e., the argument is not null or 0, then it will match the name of the terminal returned. If the boolean *use_p* is nonzero (true), then the coordinates *xp* and *yp*, given in microns, will match the placement location of the returned terminal. If both name and coordinates are given, both must match.

An empty handle (scalar 0) is returned if there is no matching physical terminal found.

(int) `CreatePhysTerminal(`*thandle*`,` *x*`,` *y*`,` *layer*`)`
As created, (electrical) terminals do not contain the data structures necessary for a corresponding terminal in the physical layout. This is fine as-is, if the user is intending to only work with a schematic, or if the terminal does not have an actual physical counterpart. However, in general one must create the physical terminal.

This function will create a new physical terminal, if one of the same name does not currently exist. The first argument can be a handle to a terminal (electrical node) or a string giving a name. In the first case, the new physical terminal is created, given the name of the electrical terminal, and the linkage established. In the second case, which does not require the existance of the electrical schematic, the physical terminal is created under the given name, and saved in the physical data. It will be linked to corresponding electrical data during association, when possible.

The *x* and *y* give the initial terminal location in the layout in microns. The layer argument can be scalar 0, which is ignored, or the name of a layer. The layer must have the Routing keyword applied. If given, this will set the layer hint for the new terminal.

The return value is 1 on success, 0 otherwise. It is not an error if the physical terminal already exists, the function will return 1 and perform no other operation in that case.

(int) `HasPhysTerminal(`*thandle*`)`
This function returns 1 if the (electrical) terminal referenced by the handle argument has a physical terminal link, 0 if no link has been assigned. On error, a value -1 is returned.

(int) `DestroyPhysTerminal(`*thandle*`)`
This will unlink and destroy the physical terminal data structure that maintains the terminal linkage into the physical layout, if any. The argument can be a handle to the corresponding electrical terminal, or to the physical terminal itself. In the latter case, the passed handle will be closed. The electrical terminal (if any) will still be valid, as will its handle if that was passed. The function returns one on success, zero if an error occurs.

(int) `GetPhysTerminalLoc(`*thandle*`, `*array*`)`
Return the layout location for the physical terminal referenced by the handle passed as an argument. The first argument can alternatively be a handle to the corresponding electrical terminal. The second argument is an array of size two or larger which will receive the x-y coordinate, in microns. The function returns one on success, zero otherwise.

(int) `SetPhysTerminalLoc(`*thandle*`, `*x*`, `*y*`)`
Set the location of the physical terminal referenced by the first argument to the layout coordinate given, in microns. The first argument can also be a handle to the corresponding electrical terminal. Generally, physical terminal locations are set by *Xic*, using extraction results. However, this may fail, requiring that the user provide a location for one or more terminals. Terminals that have been placed by the user (using this function) will by default remain fixed in the location. The function returns one on success, zero if an error occurs.

(string) `GetPhysTerminalLayer(`*thandle*`)`
Return a string containing the layer name for the physical terminal referenced by the handle passed as an argument. A handle to the corresponding electrical terminal is also accepted. Non-virtual physical terminals are associated with an object on a Routing layer. A null string is returned if there is no associated layer.

(int) `SetPhysTerminalLayer(`*thandle*`, `*layer*`)`
This function will set the associated layer hint on the physical terminal referenced by the handle passed as the first argument. A handle to the corresponding electrical terminal is also accepted. If the second argument is the name of a physical layer which has the Routing keyword set, the terminal hint layer will be set to that layer. If the second argument is a scalar 0, or a null or empty string, any existing hint layer will be removed. The function returns one on success, zero otherwise.

(int) `GetPhysTerminalGroup(`*thandle*`)`
This function will return the conductor group number to which the physical terminal referenced by the argument is assigned. A handle to the corresponding electrical terminal is also accepted. The group assignment is made during extraction and association. The return value is a non-negative integer on success, or -1 if extraction/association has not been run (or been reverted), or -2 if some error occurred.

(object_handle) `GetPhysTerminalObject(`*thandle*`)`
Return a handle to a physical object that is associated with the physical terminal referenced by the handle passed as an argument. A handle to the coresponding electrical terminal is also accepted. Physical terminals are associated with underlying conducting objects as part of the connectivity algorithm. Not all terminals have an associated object, in which case they are “virtual”. An empty handle (scalar 0) is returned in this case.

F.9.4 Physical Conductor Groups

(int) `Group()`
This function will run the grouping and device extraction algorithm on the current physical cell.

The grouping algorithm identifies the wire nets. The returned value is the number of groups used, or 0 if an error occurs. The group index extends from 0 through the number returned minus one. Group 0 is the ground group, if a ground plane layer has been defined.

(int) `GetNumberGroups()`
This returns the number of conductor groups allocated by the extraction process in the physical part of the current cell. The group index passed to other functions should be less than this value.

(int) `GetGroupBB(`*group*`,` *array*`)`
This function returns the bounding box of the conductor group whose index is passed as the first argument. The coordinates, in microns relative to the current physical cell origin, are returned in the *array*, which must have size 4 or larger. If the function succeeds, 1 is returned, otherwise 0 is returned. The saved order is L, B, R, T.

(int) `GetGroupNode(`*group*`)`
This function returns the node number from the electrical database which corresponds to the physical group index passed as the argument. If the association failed, -1 is returned.

(string) `GetGroupName(`*group*`)`
This will return a string containing a name for the group whose number is passed as the argument. The name is the name of a formal terminal attached to the group, or the net name if no formal terminal. If the group has no name, a null string is returned.

(string) `GetGroupNetName(`*group*`)`
This will return a string containing the net name for the group whose number is passed as the argument. If the group has no net name, a null string is returned.

(real) `GetGroupCapacitance(`*group*`)`
This will return the capacitance assigned to the group whose index is passed as the argument. If no capacitance has been assigned. 0 is returned.

(int) `CountGroupObjects(`*group*`)`
Return the number of physical objects that implement the group. If there is an error, such as the argument being out of range, -1 is returned.

(object_handle) `ListGroupObjects(`*group*`)`
This function returns a handle to the list of objects associated with the current physical cell which constitute the group, as found by the extraction system. These may or may not correspond to actual objects in the cell. For example, the objects returned have been processed by the `Conductor Exclude` directive, so would possibly be clipped versions of the original objects. Additionally, objects from wire-only subcells and vias that have been logically flattened during extraction will be included. Objects from flattened via instances will mave the `MergeCreated` (`0x1`) flag set, which can be tested with `GetObjectFlags`. This allows the caller to filter out redundant metal if standard vias are used, in addition to the objects, to represent the net.

The argument is the group number. The returned objects are copies, so can not be modified or selected. If an error occurs, 0 is returned.

(int) `CountGroupVias(`*group*`)`
Return the number of via instances used to implement the group, from the extraction system. This is the number of vias that would be returned by `ListGroupVias` (below). If there is an error, such as the group number argument being out of range, -1 is returned.

(object_handle) `ListGroupVias(`*group*`)`
This function returns a handle to the list of via instances associated with the current physical cell

which are used in the group, as obtained from the extraction system. This may include vias that were "promoted" due to the logical flattening of wire-only subcells during extraction. Vias in such cells are treated as if they reside in their parent cells, recursively.

The argument is the group number. The via instances are copies, so can not be modified or selected. If an error occurs, 0 is returned.

(int) `CountGroupDevContacts(`*group*`)`
This function returns a count of the number of device contacts which are assigned to the conductor group whose index is passed as the argument. If an error occurs, -1 is returned.

(dev_contact_handle) `ListGroupDevContacts(`*group*`)`
This function returns a handle to the list of device contacts which are assigned to the conductor group whose index is passed as the argument. If an error occurs, 0 is returned.

(int) `CountGroupSubcContacts(`*group*`)`
This function returns a count of subcircuit contacts associated with the group index passed as the argument. If an error occurs, -1 is returned.

(subc_contact_handle) `ListGroupSubcContacts(`*group*`)`
This function returns a handle to a list of subcircuit contacts associated with the group index passed as the argument. If an error occurs, 0 is returned.

(int) `CountGroupTerminals(`*group*`)`
Return a count of cell connection terminals associated with the group number passed as an argument. If an error occurs, -1 is returned.

(terminal_handle) `ListGroupTerminals(`*group*`)`
This will return a handle to a list of formal terminals associated with the group number passed as an argument. If an error occurs, 0 is returned. If the group contains no formal terminals, the list will be empty.

(stringlist_handle) `ListGroupTerminalNames(`*group*`)`
This function returns a list of names of the cell connection terminals assigned to the conductor group whose index is passed as the argument. If an error occurs, 0 is returned. If the group contains no cell connection terminals, the list will be empty.

(int) `CountGroupPhysTerminals(`*group*`)`
Return a count of the physical terminal descriptors from the physical cell that are associated with the group number given.

(physterm_handle) `ListGroupPhysTerminals(`*group*`)`
Return a handle to a list of the physical terminal descriptors from the physical cell that are associated with the group number given.

F.9.5 Physical Devices

(device_handle) `ListPhysDevs(`*name*`, `*pref*`, `*indices*`, `*area_array*`)`
This function returns a handle to a list of devices extracted from the physical part of the current cell. The first two arguments are strings which match the `Name` and `Prefix` fields from the technology file Device block of the device to list. Either or both of these arguments can be null or empty, in which case no devices are excluded by the comparison, i.e., such values act as wildcards.

The third argument is a string providing a list of device indices, or ranges of indices, to allow. These are integers that are unique to each instance of a device type in a cell. If this argument is null or empty, all indices will be returned. Each token in the string is an integer (e.g., "2"), or range of integers (e.g., "1-4"), using the hyphen (minus sign) to separate the minimum and maximum index to include. The tokens are separated by white space and/or commas. For example, "1,3-5,7,9-12".

The final argument, if not 0, is an array of size four or larger containing rectangle coordinates, in microns, in order L,B,R,T. If 0 is passed for this argument, the entire cell is searched for devices. Otherwise, only the area provided will be searched.

On success, a handle is returned, otherwise 0 is returned. The handle can be used in the functions that take a device handle as an argument. This is *not* an object handle. The returned device handle can be manipulated with the generic handle functions, and like other handles should be iterated through or explicitly closed when no longer needed.

(string) `GetPdevName(`*device_handle*`)`
This function returns a string containing the name of the device referenced by the handle. The name string is composed of the `Name` field for the device (from the Device Block), followed by an underscore, followed by the device index number. If the handle is defunct or some other error occurs, a null string is returned.

(int) `GetPdevIndex(`*device_handle*`)`
This function returns the index of the device referenced by the handle passed as an argument. The index is an integer which is unique among the devices of a given type. If the handle is defunct or an error occurs, -1 is returned.

(object_handle) `GetPdevDual(`*device_handle*`)`
This function returns an object handle which references the dual device in the electrical database to the physical device referenced by the argument. If association failed for the device, 0 is returned. The dual device is a subcell obtained from the device library.

(int) `GetPdevBB(`*device_handle*`,` *array*`)`
This function obtains the bounding box of the device referenced by the first argument. The coordinates, in microns using the origin of the current physical cell, are returned in the *array*, which must have size 4 or larger. If the function succeeds, 1 is returned, otherwise the returned value is 0. The saved order is L, B, R, T.

(real) `GetPdevMeasure(`*device_handle*`,` *mname*`)`
This function returns a device parameter corresponding to a `Measure` line given in the Device block for the device referenced by the first argument. The second argument is a string giving the name from a `Measure` line. The returned value is the measured parameter, or 0 if there was an error.

(stringlist_handle) `ListPdevMeasures(`*device_handle*`)`
This function returns a string list handle corresponding to a list of the names associated with `Measure` lines in the Device block for the device referenced by the handle. These are the names that can be passed to `GetPdevMeasure` to perform the measurement. If an error occurs, 0 is returned.

(dev_contact_handle) `ListPdevContacts(`*device_handle*`)`
This function returns a handle to a list of contact descriptors for the device referenced by the argument. The returned handle can be passed to the functions below to obtain information about the device contacts. If there is an error, 0 is returned. The returned handle can be manipulated with the generic handle functions, and like other handles should be iterated through or closed explicitly when no longer needed.

(string) `GetPdevContactName(`*dev_contact_handle*`)`
This function returns the name string of the contact referenced by the argument. Contact names

are assigned in the Device block for the device containing the contact. If an error occurs, a null string is returned.

(int) `GetPdevContactBB(`*dev_contact_handle*`,` *array*`)`
This function returns the bounding box of the contact referenced by the first argument. The coordinates, in microns relative to the origin of the physical current cell, are returned in the *array*, which must have size 4 or larger. If the operation is successful, 1 is returned, otherwise 0 is returned.

(int) `GetPdevContactGroup(`*dev_contact_handle*`)`
This function returns the conductor group index to which the contact referenced by the argument is assigned. If there is an error, -1 is returned.

(string) `GetPdevContactLayer(`*dev_contact_handle*`)`
This function returns the name string of the layer to which the contact referenced by the argument is assigned. All contacts are assigned to layers which have the Conductor attribute. If there is an error, a null string is returned.

(device_handle) `GetPdevContactDev(`*dev_contact_handle*`)`
This function returns a handle to the device containing the contact referenced by the argument. If an error occurs, 0 is returned. The returned handle should be closed (for example, with the `Close` function) when no longer needed.

(string) `GetPdevContactDevName(`*dev_contact_handle*`)`
This function returns the name of the device containing the contact referenced by the argument. A null string is returned on error.

(int) `GetPdevContactDevIndex(`*dev_contact_handle*`)`
This returns the index number of the device to which the contact, referenced by the passed handle, is associated. Each device of a given type has an index number assigned, which is unique in the containing cell. On error, -1 is returned. A valid index is 0 or larger.

F.9.6 Physical Subcircuits

(subckt_handle) `ListPhysSubckts(`*name*`,` *index*`,` *l*`,` *b*`,` *r*`,` *t*`)`
This function returns a handle to a list of subcircuits from the physical part of the current cell. Subcircuits are subcells which contain devices or sub-subcells that contain devices. Subcells that contain only wire are typically not saved internally as subcircuits. The first argument is a string name which will match the returned subcircuits. If this argument is null or empty, then this test will not exclude any subcircuits to be returned. The second argument is the index number of the subcircuit to be returned. If the value is -1, subcells with any index will be returned. The remaining four values define a rectangular area, given in microns relative to the current physical cell origin, where subcircuits will be searched for. If all four values are 0, the entire cell will be searched. The returned handle references subcircuits, and is distinct from device handles and object handles. The handle can be passed to the generic handle functions, and like other handles should be iterated through or closed when no longer needed. The function returns 0 if an error occurs.

(string) `GetPscName(`*subckt_handle*`)`
This function returns the cell name corresponding to the subcircuit referenced by the handle. if an error occurs, a null string is returned.

(int) `GetPscIndex(`*subckt_handle*`)`
This function returns the index of the subcircuit referenced by the argument. The index is a zero-based sequence for each subcircuit master. If an error occurs, -1 is returned.

(int) `GetPscIdNum(`*subckt_handle*`)`
This function returns the ID number of the subcircuit referenced by the argument. The ID number is unique among all instances in the parent cell. If an error occurs, -1 is returned.

(string) `GetPscInstName(`*subckt_handle*`)`
This function returns an instance name corresponding to the subcircuit instance referenced by the handle. This is the cell name, followed by an underscore, followed by the index number. if an error occurs, a null string is returned.

(object_handle) `GetPscDual(`*subckt_handle*`)`
This function returns an object handle which references the subcell in the electrical database which is the dual of the physical subcircuit referenced by the argument. If the association fails, 0 is returned.

(int) `GetPscBB(`*subckt_handle*`,` *array*`)`
This function returns the bounding box of the subcircuit referenced by the first argument. The coordinates, in microns relative to the origin of the current physical cell, are returned in the array, which must have size 4 or larger. If the operation succeeds, 1 is returned, otherwise 0 is returned.

(int) `GetPscLoc(`*subckt_handle*`,` *array*`)`
This returns the instance placement location, in microns, in the array passed as a second argument. The array must have size two or larger. On success, the function returns 1, and the array location 0 will contain the X value, and the 1 location will contain the Y value. Zero is returned on error, with the array values undefined.

(int) `GetPscTransform(`*subckt_handle*`,` *type*`,` *array*`)`
This function returns a string describing the instance orientation. There are presently three format types, specified by the second argument. If this argument is zero, then the *Xic* transformation string is returned. This is the same CIF-like encoding as used for the current transformation in the status line of *Xic*. In this case the third argument is ignored and can be zero.

If the second argument is one, the return will be a Cadence DEF orientation code. In addition, if an array of size two or larger is passed as a third argument, the values will be filled in with the X and Y origin correction values implied by the transformation. In a DEF transformation, the lower left corner position of the bounding box is invariant, implying that there is an additional translation after rotation/mirroring to enforce this. Pass 0 for this argument if these values aren't needed.

In DEF, there is no support for 45, 135, 225, and 315 rotations, a null string is returned in these cases. Magnification is ignored.

If the second argument is any other value, the OpenAccess strings are returned, otherwise all is as for DEF.

The following table lists equivalent orientation codes for DEF, OpenAccess, and *Xic*. The **Origin** column indicates the position of the original lower-left corner after the operation.

LEF/DEF	OpenAccess	*Xic*	Origin
N	R0	R0	LL
W	R90	R90	LR
S	R180	R180	UR
E	R270	R270	UL
FN	MY	MX	LR
FW	MX90	R270MY	LL
FS	MX	MY	UL
FE	MY90	R90MX	UR

(subc_contact_handle) `ListPscContacts(`*subckt_handle*`)`
This function returns a handle to a list of subcircuit contacts associated with the subcircuit referenced by the handle. The returned handle is a distinct type, in particular subcircuit contacts are different from device contacts. The return handle can be used with the functions which query information about subcircuit contacts, or with the generic handle functions. If an error occurs, this function returns 0.

(int) `IsPscContactIgnorable(`*subc_contact_handle*`)`
If the subcircuit associated with the contact referenced from the argument is flattened or ignored, return 1. Otherwise 0 is returned. When 1 is returned, the contact can usually be skipped in listings.

(string) `GetPscContactName(`*subc_contact_handle*`)`
This function returns a name string, if available, from the subcircuit contact referenced by the argument. If the subcircuit does not provide a name, the returned string will be a number giving the subcircuit group contacted. A null string is returned on error.

(int) `GetPscContactGroup(`*subckt_contact_handle*`)`
This function returns the group index in the current cell corresponding to the subcircuit contact referenced by the argument. If an error occurs, this function returns -1.

(int) `GetPscContactSubcGroup(`*subckt_contact_handle*`)`
This function returns the group index in the subcircuit associated with the subcircuit contact referenced by the argument. On error, the function returns -1.

(subckt_handle) `GetPscContactSubc(`*subckt_contact_handle*`)`
This function returns a handle to the subcircuit which is associated with the subcircuit contact referenced by the argument. On error, the function return 0.

(string) `GetPscContactSubcName(`*subc_contact_handle*`)`
This function returns a string containing the name of the subcircuit associated with the contact referenced by the argument. A null string is returned on error.

(int) `GetPscContactSubcIndex(`*subc_contact_handle*`)`
This function returns the index of the subcircuit associated with the contact referenced by the argument. Each subcircuit of a given kind has an index number that is unique in the containing cell. On error, -1 is returned. Valid index values are 0 and larger.

(int) `GetPscContactSubcIdNum(`*subc_contact_handle*`)`
This function returns the ID number of the subcircuit associated with the contact referenced by the argument. Each subcircuit has an ID number that is unique in the containing cell. On error, -1 is returned. Valid index values are 0 and larger.

(string) `GetPscContactSubcInstName(`*subc_contact_handle*`)`
This function returns a string containing an instance name of the subcircuit associated with the contact referenced by the argument. The instance name consists of the cell name followed by an underscore, which is followed by the index. A null string is returned on error.

F.9.7 Electrical Devices

(stringlist_handle) `ListElecDevs(`*regex*`)`
This function returns a handle to a list of strings containing device names from the electrical database. The names correspond to devices used in the current circuit. The argument is a regular expression used to filter the device names. If the argument is null or empty, all devices are listed. This function returns 0 on error.

(int) `SetEdevProperty(`*devname*`, `*prpty*`, `*string*`)`
This function is used to set property values of electrical devices and mutual inductors. It is equivalent to the `Set` command, or the keyboard **!set** command, with the *devname.prpty* syntax. The first argument is the name of a device in the current circuit. This is the value of a name property for some device. The second argument is a string giving the property type to set or modify. The possible strings are prefixes of "`name`", "`model`", "`value`", "`param`", "`other`", and "`nophys`". The single character string "`n`" implies name, and (additionally) "`y`" implies nophys. If the string is unrecognized, the property type defaults to other. If the device is a mutual inductor, only the name and value properties can be applied. The final argument is a string containing the body of the property. If the string is null or empty, the property is removed (or reset to the default in the case of the name property). The function returns 1 on success, 0 otherwise.

(string) `GetEdevProperty(`*devname*`, `*prpty*`)`
This function returns a string containing the text of the specified property for the given device. The two arguments have the same format and interpretation as the first two arguments of `SetEdevProperty`, i.e., the device name and property name. The return value is a string containing the text for that property. If the device or property does not exist or some other error occurs, a null string is returned.

(object_handle) `GetEdevObj(`*devname*`)`
This function returns a handle to the electrical subcell from the device library corresponding to the given device name. If an error occurs, 0 is returned.

F.9.8 Resistance/Inductance Extraction

(int) `ExtractRL(`*conductor_zoidlist*`, `*layername*`, `*r_or_l*`, `*array*`, `*term*`, ...)`
This will use the square-counting system to estimate the resistance or inductance of a conducting object with respect to two or more terminals. The first argument is a trapezoid list representing a single conducting area, on the layer given in the second argument. The layer keywords set electrical parameters used in the estimation.

For Resistance:
The Rsh layer keyword gives the ohms-per-square of the material. If not set, the value is computed from Rho or Sigma and Thickness if these are set. If these keywords are also not given, a value of 1.0 is assumed.

For Inductance:
The `Tline` keyword supplies the appropriate parameters. In this case, the material is assumed to be over a ground plane covered by dielectric.

The third argument is a boolean which if nonzero indicates inductance estimation, and zero indicates resistance estimation.

The fourth argument is an array which will hold the return values, which will be resized if necessary. The zeroth component of the array gives the number of returned values, which are returned in the rest of the array. If there are two terminals, the number of returned values is 1. For more than two terminals, the number of returned values is $n*(n-1)/2$, where n is the number of terminals. The values are the effective two-terminal decomposition for terminals i,j (i `!=` j) in the order, e.g., for $n = 4$; 01, 02, 03, 12, 13, 23.

The following arguments are trapezoid lists representing the terminals. Arguments that are not trapezoid lists will be ignored. There must be at least two terminals passed. Terminal areas should be spatially disjoint, and in the computation, the terminal areas are clipped by the conductor area. Terminals are assigned numbers in left-to-right order.

The algorithm is most efficient if all coordinates are on some grid. This provide for efficient tiling of the structure.

Structures that require a very large number of tiles may require excessive time and memory to compute, and/or suffer from a loss of accuracy. The approximate threshold is 10^5 tiling squares. Non-Manhattan shapes have strict internal limiting of tile count. Manhattan structures can require an arbitrarily large number of tiles, thus the potential for resource overuse.

The return value is always 1. The function will fail (terminating the script) if an error is encountered.

(int) `ExtractNetResistance(`*net_handle*`,` *spicefile*`,` *array*`,` *term*`, ...)`

This function will extract resistance of a conductor net, taking into account multiple conducting layers connected by vias. The resistance decomposition of each conducting object and its vias and/or terminals is computed using the algorithm used by the `ExtractRL` function. The resistance of the connected network is then computed, with respect to the terminals specified.

The first argument is a handle to a list of objects as returned from `FindPath` or `FindPathOfGroup`.

The second argument is a string giving a file name, which will contain a generated SPICE listing representing the extracted resistor network. In the SPICE file, each terminal and each via are assigned node numbers. A comment indicates the range of numbers used for terminals. If this argument is 0 (NULL) or an empty string, no SPICE file is written.

The third argument is an array which will hold the return values, which will be resized if necessary. The zeroth component of the array gives the number of returned values, which are returned in the rest of the array. If there are two terminals, the number of returned values is 1. For more than two terminals, the number of returned values is $n*(n-1)/2$, where n is the number of terminals. The values are the effective two-terminal decomposition for terminals i,j (i `!=` j) in the order, e.g., for $n = 4$; 01, 02, 03, 12, 13, 23.

The following arguments are trapezoid lists representing the terminals. Arguments that are not trapezoid lists will be ignored. There must be at least two terminals passed. Terminal areas should be spatially disjoint, and in the computation, the terminal areas are clipped by the conductor area. Terminals are assigned numbers in left-to-right order.

The return value is always 1. The function will fail (terminating the script) if an error is encountered.

F.10 Schematic Editor Functions

F.10.1 Output Generation

(int) `Connect(`*for_spice*`)`

This function establishes the circuit connectivity for the current hierarchy. If the boolean *for_spice* is false, then devices with the nophys property set are ignored, and the netlist will have the "shorted" nophys devices shorted out. This is appropriate for LVS and other extraction system operations.

If *for_spice* is true, the nophys devices are included, and not shorted. This applies when generating output for SPICE simulation.

The function returns 1 on success, 0 otherwise. If the schematic is already processed and current, the function will return immediately. The schematic is implicitly processed before most internal operations that make use of the schematic, so it is unlikely that the user will need to call this function.

(int) `ToSpice(`*spicefile*`)`

This function will dump a SPICE file from the current cell to a file of the given name. If the argument is null or an empty string, the name will be that of the current cell with a "`.cir`" suffix. Any existing file of the same name will be moved, and given a "`.bak`" extension. The return value is 1 on success, 0 otherwise.

F.10.2 Electrical Nodes

(int) `IncludeNoPhys(`*flag*`)`

This sets an internal mode which applies to the other functions in this group. If the boolean *flag* argument is nonzero, devices with the nophys property set will be considered when generating the connectivity and node mapping structures. This has relevance when a device has the shorted option to nophys set, as such devices will be considered as normal devices with the flag set. If the flag is unset, these devices will be taken as short circuits, which of course alters the node assignments.

Internally, the extraction functions always take these devices as shorted, and they are otherwise ignored. When generating a SPICE file during simulation or with other commands in the side menu, these devices are included as normal devices. The present state of the netlist data structures will reflect the state of the last operation.

Setting this flag will cause rebuilding of the data structures to the requested state if necessary when one of the functions in this section is called. This persists until some other function, such as an extraction or SPICE listing function is called, at which time the internal state of the flag may change. Thus, this function may need to be called repeatedly ahead of the functions in this section.

The return value is the previous value of the internal flag.

(int) `GetNumberNodes()`

Return the size of the internal node map. The internal node numbers range from 0 up to but not including this value. The return value is 0 on error or if the cell is empty.

(int) `SetNodeName(`*node*`, `*name*`)`

This function associates the string *name* with the node number given in the first argument. This affects the electrical database, and is equivalent to setting a node name with the node mapping facility available in the side menu in electrical mode. Netlist output will use the given string name rather than a default name, however if the existing default name matches a global node name, the user-supplied name will be ignored. If the name given is null or empty, any existing given name is deleted, and netlist output will use the node number. The function returns 1 on success, 0 otherwise.

(string) `GetNodeName(`*node*`)`
This function returns a string name for the given node number. If a name has been given for that node, the name is returned, otherwise an internally generated default name is returned. If the operation fails, a null string is returned.

(int) `GetNodeNumber(`*name*`)`
This function returns the node number corresponding to the name string passed as an argument. If no mapping to the string is found, -1 is returned.

(int) `GetNodeGroup(`*node*`)`
This function returns the group index in the physical cell that corresponds to the given node number. On error, -1 is returned.

(terminal_handle) `ListNodePins(`*node*`)`
Note: This and `ListNodeContacts` replace `ListNodeTerminals`, which was removed in 4.2.12.

Return a handle to the list of cell connection terminals bound to the internal node number supplied as the argument. There probably will be at most one such connection.

(terminal_handle) `ListNodeContacts(`*node*`)`
Note: This and `ListNodePins` replace `ListNodeTerminals`, which was removed in 4.2.12.

Return a handle to a list of device and subcircuit connection terminals bound to the specified node.

(object_handle) `GetNodeContactInstance(`*terminal_handle*`)`
For a handle to an instance contact, such as returned from `ListNodeContacts`, this function will return a handle to the device or subcircuit instance that provides the contact.

(stringlist_handle) `ListNodePinNames(`*node*`)`
Note: This and `ListNodeContactNames` replace `ListNodeTerminalNames`, which was removed in 4.2.12.

Return a list of cell connection terminal names that connect to the given node. There is likely at most one cell connection per node.

(stringlist_handle) `ListNodeContactNames(`*node*`)`
Note: This and `ListNodePinNames` replace `ListNodeTerminalNames`, which was removed in 4.2.12.

Return a list of device and subcircuit contact names that connect to the given node.

F.10.3 Symbolic Mode

(int) `IsShowSymbolic()`
This function will return 1 if the current cell is being displayed in symbolic form in the main window, 0 otherwise. The return is always 0 in physical mode.

(int) `ShowSymbolic(`*show*`)`
This will set symbolic mode of the current cell, and display the symbolic representation, if possible, in the main window. The effect is similar to the effect of pressing or un-pressing the **symbl** button in the electrical side menu. The function call must be made in electrical display mode. When symbolic mode is asserted, by passing a boolean true argument, the current cell will be displayed in symbolic mode, unless the **No Top Symbolic** button in the **Main Window** sub-menu of the

Attributes Menu is pressed. The return value is 1 on success, 0 if some error occurred, with an error message likely available from `GetError`.

(int) `SetSymbolicFast(`*symb*`)`

This will enable or disable symbolic mode of the current cell. It differs from `ShowSymbolic` in two ways. First, it applies only to cells with a symbolic representation, meaning that it has a symbolic form which may or may not be visible. Second, it will change the status of a flag in the cell, but there will be no updating of the screen or other internal things (such as undo logging). The caller must reset to the original state before a screen redisplay or any major operation. This is much faster than calling `ShowSymbolic`, and can be used when making quick changes to a cell.

The return value is 1 if the current cell was previously actively symbolic, 0 otherwise. In physical mode the return value is always 0 and the function has no effect.

(int) `MakeSymbolic()`

This will create a very simple symbolic representation of the electrical view of the current cell, consisting of a box with a name label, and wire stubs containing the terminals. Any existing symbolic representation will be overwritten (but the operation can be undone). In electrical mode, symbolic mode will be asserted.

On success, 1 is returned, 0 otherwise.

This page intentionally left blank.

Appendix G

The FileTool Utility

G.1 Introduction

The *FileTool* is a command-line program for analysis and manipulation of layout files. Although the *FileTool* originated as a separate stand-alone application that made use of *Xic* technology, the current version is a polymorphism of the *Xic* executable. There are two ways to access the *FileTool*:

1. Copy or symbolically link one the `xic` executable file (or the `xic.exe` file under Windows) to a new link or file named "`filetool`" (or "`filetool.exe`" under Windows). You now have a *FileTool* program that behaves in all respects as described in this documentation.

   Under Unix/Linux/macOS, the best way is to use a symbolic link. For example, in the same directory as the `xic` executable, become root and type (for example)

   ```
   ln -s xic filetool
   ```

   This will symbolically link the `xic` binary executable to the `filetool` name, without actually copying the file. If the `xic` file is replaced for an update, the link will automatically access the new executable.

   This is **not** automatically done when the programs are installed. The user must intervene to obtain a `filetool` executable target.

   Under Windows, there are no symbolic links, so the file must actually be copied. Thus, after an update, the copy operation should be repeated, to obtain any updates that relate to the *FileTool*.

2. One can also effectively run the *FileTool* directly from *Xic* with, for example,

   `xic -F` *filetool_args...*

   The `-F` must be the first argument, and all arguments that follow are interpreted as *FileTool* arguments. The program will behave in all respects as if started under the name "`filetool`".

The *FileTool* can be incorporated in the user's automation scripts to implement perhaps complicated manipulations on layout files, or as an aid to understanding content and diagnosing problems with layout files, or as a general purpose utility. Here are some of the tasks that the *FileTool* can perform:

- Print information about a layout file: statistics, layers used, top-level cell, etc.

- Translate layout files, or parts of layout files, to a different format (CIF, CGX, GDSII, OASIS are supported), or to an ASCII text representation.
- When writing, many different translation modes are available: layer filtering and aliasing, cell name mapping operations, windowing with or without clipping, flattening, scaling, empty cell removal.
- Compare two layout files, listing the differences.
- Split a layout file into multiple files, each representing a portion of the original layout.
- Combine cells from multiple layout files into a single file.
- Generate or process assemble scripts as used by the *Xic* **!assemble** command.

When started, none of the *Xic* startup or technology files are read. Instead, a file named "`.filetoolrc`" will be read, if it can be found in the current directory of the user's home directory. This is a script file, like the `.xicstart` file, however the only function likely to be useful is the `Set` function, which sets variables. Variables can also be set from the *FileTool* command line, but the `.filetoolrc` file can be used to set variables that are almost always needed, such as favorite OASIS flags when working with OASIS files.

The file formats supported by the *FileTool* are:

GDSII
The industry standard stream format. Any release level is supported for input. For output, the default release level is 7, but this can be set to earlier levels. Compressed (gzipped) GDSII files can be read or written.

OASIS
The emerging standard, which provides more compact data files than GDSII. Any conforming OASIS file can be read as input. A number of options affect OASIS output.

CGX
A compact data representation developed by Whiteley Research Inc. Compressed (gzipped) CGX files can be read or written.

CIF
The obsolete but still used CIF format. Any known dialect should work as input. The output dialect can be selected via options.

Input files can be any of these file types, the format is recognized automatically. Output files can also be any of these file types, but the format is specified by the extension of the file name.

The operations can be saved to a script file, or read from a script file. The script file format is the same as used by the **!assemble** command in *Xic*, thus scripts generated by the *FileTool* can be executed in *Xic*.

G.2 Command Line Options

If the *FileTool* is executed without arguments, a synopsis of available command line options is printed. Otherwise, the arguments are given in one of the following forms.

filetool [-set *var*[=*value*] ...]
  -eval *script_file_to_read* |
  -info *layout_file* [*flags*] |
  -text *layout_file* [*text_opts*] |
  -comp *comp_args* |
  -split *split_args* |
  -cfile *cfile_args* |
  translate_args

The `-set` option is used to set internal variables, which have relevance in the modes indicated by the other main options.

The `-eval` option is used to execute an assemble script.

The `-info` option is used to obtain information and statistics about a layout file.

The `-text` option will translate all or part of a layout file to an ASCII text representation.

The `-comp` option will set up a comparison of two layout files, recording differences.

The `-split` option is used to write multiple layout files corresponding to regions in a large layout.

The `-cfile` option is used to write a Cell Hierarchy Digest (CHD) file from a layout file, similar to the **Save** button in the **Cell Hierarchy Digests** panel.

Otherwise, the given options are expected to provide directives similar in logic to that of an assembly script.

G.3 FileTool: Setting Variables

There are a number of internal variables which control various properties of the file readers/writers, translation modes, etc. These are the same variables as used in *Xic*. In some cases, these variables are overridden by command line options, but in cases where no applicable option exists, these variables can be set to provide the desired effect. Variables can also be set in the `.filetoolrc` file. Variables set from the command line will override settings in the `.filetoolrc` file.

The `-set` options must appear first on the command line, and unlike the other main directives, can appear ahead of the other directives. These are optional.

The format can take two forms: either a single `-set` option followed by a quoted list of *name=value* pairs:

-set "*name1=value1 name2 name3=value3 ...*"

or, each *name=value* pair can have its own "`-set`":

-set *name1=value1* -set *name2* -set *name3=value3*

Note that the value part is optional, for boolean variables. The token following each "`-set`" must not contain white space, or be quoted if it contains white space, e.g.,

-set "*name = value*"

is legitimate.

The following variables have relevance to operations that are available through the *FileTool*.

In addition to the variables listed in the table, which are *Xic* variables, there is one special boolean variable recognized:

`timedbg`[=*filename*]

If set, run times for various operations are printed, similar to enabling the **!timedbg** feature in *Xic*. If set to a value, the value is taken as a path to a file for the timing messages.

Database Setup	
DatabaseResolution	
Symbol Path	
Path NoReadExclusive	AddToBack
Conversion - General	
ChdFailOnUnresolved MultiMapOk UnknownGdsLayerBase	UnknownGdsDatatype NoStrictCellnames
Conversion - Import and Conversion Commands	
AutoRename NoOverwritePhys NoOverwriteElec NoOverwriteLibCells NoCheckEmpties NoReadLabels MergeInput NoPolyCheck DupCheckMode LayerList UseLayerList LayerAlias	UseLayerAlias InToLower InToUpper InUseAlias InCellNamePrefix InCellNameSuffix NoMapDatatypes CifLayerMode OasReadNoChecksum OasPrintNoWrap OasPrintOffset
Conversion - Export Commands	
StripForExport KeepLibMasters SkipInvisible KeepBadArchive NoCompressContext RefCellAutoRename UseCellTab SkipOverrideCells OutToLower OutToUpper OutUseAlias OutCellNamePrefix OutCellNameSuffix CifOutStyle CifOutExtensions	CifAddBBox GdsOutLevel GdsMunit NoGdsMapOk OasWriteCompressed OasWriteNameTab OasWriteRep OasWriteChecksum OasWriteNoTrapezoids OasWriteWireToBox OasWriteRndWireToPoly OasWriteNoGCDcheck OasWriteUseFastSort OasWritePrptyMask
Geometry	
JoinMaxPolyVerts JoinMaxPolyGroup JoinMaxPolyQueue	JoinBreakClean PartitionSize

G.4 FileTool: Assemble Script File Evaluation

Assemble script files can be produced by *Xic*, and contain a specification for complicated operations on layout files, such as merging several files into a single output file, while creating a new top-level cell to contain instances of the cells read from input. These files can be evaluated with the *FileTool*.

The command is of the form

`filetool` [`-set` *variables*] `-eval` *script_file*

The FileTool will read and execute the script, reading input and generating output as per the directives in the script file.

The script file format is described in 19.2.3.

G.5 FileTool: Obtaining File Information

In this mode, the *FileTool* will read a layout file, and print useful information about the file. The command line for this mode is

`filetool` [`-set` *variables*] `-info` *filename* [*flags*]

It is unlikely that the `-set` variables will be used with this option, though the layer filtering options may apply on occasion.

The output format and *flags* are identical to those of the *Xic* **!fileinfo** command (19.14.1).

G.6 FileTool: ASCII Text Representation of Layout Files

The supported file formats other than CIF are binary, and thus the content is not easy to decipher. This mode of the *FileTool* will convert records from a layout file into an ASCII representation. This may be valuable for identifying problems in the file or understanding file organization and content.

For this mode, the command takes the form:

`filetool` [`-set` *variables*] `-text` *layout_file* [`-o` *output_file*] [*start*[-*end*]] [`-c` *cells*] [`-r` *recs*]

Following the layout file path, there are optional arguments.

`-o` *output_file*
: If this is given, the text output will be placed in the supplied file name. Without this option given, text output is to the standard output.

  The remaining arguments control the range of text conversion. Without these options, the entire file will be written as ASCII text. For all but tiny layout files, the user will probably want to limit the size of the output.

[*start*[-*end*]]
: The *start* and *end* are file offsets, which can be given in decimal or "0x" hex form. Printing will start with the first record with offset greater than or equal to *start*. If *end* is given, the last record printed will be at most the record containing this offset. If both numbers are given, they must be separated by a '-' with no white space.

`-c` *cells*
: This options supplies a count, indicating the number of cell definitions that will be printed. If the count is 0, and *start* is also given, the records from *start* to the end of the cell definition will be printed.

`-r` *recs*
This provides a count of the number of records to print. Printing will stop after the indicated number of records have been output.

Printing will start at the beginning of the file or the *start* record if given, and will end at the end of file or the point at which the first end condition is satisfied.

There are two variables which may be of interest when using this mode. These can be set with `-set` options ahead of the `-text` argument.

OasPrintNoWrap
Value: boolean
This applies when converting OASIS input to ASCII text. When set, the text output for a single record will occupy one (arbitrarily long) line. When not set, lines are broken and continued with indentation.

OasPrintOffset
Value: boolean
This applies when converting OASIS input to ASCII text. When set, the first token for each record output gives the offset in the file or containing CBLOCK. When not set, file offsets are not printed.

G.7 FileTool: Layout File Comparison

This mode compares the geometry and instance placements in cells from two cell hierarchies, usually from different files. The results are written to a log file.

The command line format for this mode is

`filetool` [`-set` *variables*] `-comp` *comp_args*

The operations and arguments are identical to those of the *Xic* **!compare** command (19.14.3). This includes the operations involving Cell Hierarchy Digests (CHDs) and in-memory hierarchies, provided that those have been created by script functions in the `.filetoolrc` file. However, it is most likely that from the *FileTool*, the sources will always be on-disk layout files.

G.8 FileTool: Layout File Splitting

The *FileTool* can be used to split a large layout file into a collection of smaller layout files.

For splitting, the command line takes the form:

`filetool` [`-set` *variables*] `-split` *split_args*

This mode will write output files corresponding to the partitions of a square grid logically covering all or part of a specified cell in a given layout file. The output files contain physical data only. These files can be flat or hierarchical.

The operations and *split_args* are identical to those of the *Xic* **!splwrite** command (19.2.4).

G.9 FileTool: CHD File Generation

The *FileTool* can be used to generate a Cell Hierarchy Digest (CHD) file. The file format is the same as produced with the **Save** button in the **Cell Hierarchy Digests** panel. The CHD file can subsequently be read into the **Cell Hierarchy Digests** panel with the **Add** button.

The command line takes the form:

`filetool` [`-set` *variables*] `-cfile -i` *layout_file* `-o` *chd_file* [`-g`] [`-c`]

If the `-g` option is given, geometry records will be included in the file. These records are effectively a concatenation of a Cell Geometry Digest file representation. Layer filtering can be employed to specify layers to include.

The resulting file is a highly compact but easily random-accessible representation of the layout file.

Future releases of *Xic* will make use of these files in creative ways, stay tuned.

The `-c` option will skip use of compression when creating the file. Files produced with this option (and without geometry) should be compatible with *Xic* release 3.2.16 and earlier, which did not support compression. If backward compatibility is not needed, this option should not be used.

G.10 FileTool: Layout File Merging and Translation

The *FileTool* can take a list of arguments which correspond logically to the keywords of an assembly specification script. The argument list begins after any `-set` variables present.

This automates reading of cells from archives, subsequent processing, and writing to a new archive file. It provides the capabilities of the **Conversion** panel in the **Convert Menu** in *Xic*, such as format translation, windowing, and flattening. Additionally, multiple input files and cells can be processed and merged into a larger archive, on-the-fly or by using a Cell Hierarchy Digest (CHD) so as to avoid memory limitations. Cell definitions for the read and possibly modified cells are streamed into the output file, and the output file can contain a new top-level cell in which the cells read are instantiated. The input and output can be any of the supported archive formats (CGX, CIF, GDSII, OASIS), in any combination.

The same operations can be controlled by a specification script file, the path to which is given as the argument following "`-eval`". The script uses a language which will be described. This supplies the output file name and the description of the top-level cell (if any), the files to be used as input, the cells to extract from these files, and the operations to perform. It is a simple text file, prepared by the user, containing a number of keywords with values. The specification script can also be obtained from the **Assemble** command in the **Convert Menu**, which is a graphical front-end to the **!assemble** command in *Xic*.

Alternatively, the argument list can consist of a series of option tokens and values. These are logically almost equivalent to the language of the specification file. This gives the user the option to enter job descriptions entirely from the command line. These command-line options start with a '`-`' character.

Only physical data are read, electrical data will be stripped in output. A log file is produced when the command is run. If not specified with a `LogFile`/`-log` directive, "`filetool.log`" and is written in the current directory. The log file contains warning and error messages emitted by the readers during file processing, and should be consulted if a problem occurs.

The details of the file format and corresponding command line options are provided in the description of the **!assemble** command.

This page intentionally left blank.

Appendix H

The *XicTools* Accessories

The *XicTools* accessories are programs provided in the optional accessories distribution file. These are open-source programs, and the source code distribution can be found in the free software archive of `wrcad.com`.

The programs in the accessories distributions are the following.

`mozy`
: A stand-alone help/www browser. This can be used to browse the *Xic* and *WRspice* help databases, or general HTML and image files. It supports HTML-3.2 and a few 4.0 features, so is no longer much good as a web browser, but it works well as a help system and viewer.

`xeditor`
: A stand-alone text editor window, as descriptd in 3.13.2. This is one polymorph of a widget that can be configured as a file browser or email client as well.

`httpget`
: A program for retrieving files served from a remote FTP or HTTP server.

`hlp2html`
: A program for creating standard HTML files from help database (`.hlp`) files.

`hlp2latex`
: A utility for extracting LaTeX versions of the help text saved in (`.hlp`) files.

`hlpsrv`
: A bridge program to allow access to a help database through a web server.

`fcpp`
: A post-processor for *FastCap* and *FasterCap* (from `FastFieldSolvers.com`) output, which finds and prints the capacitance values.

`lstpack` and `lstunpack`
: Utilities to convert *FastCap* input files between the packed (single file) and unpacked (multiple file) formats. The Whiteley Research release of *FastCap* and *FasterCap* from `FastFieldSolvers.com` understand the packed format, other versions do not.

These tools and the supporting libraries are provided in the hope that they may be useful, under the GNU Library General Public License. This is open-source software, with no guarantees whatsoever, use at your own risk.

The graphics support library was originally written as a toolkit-independent layer between the application and the GUI. It is used in all graphical *XicTools* programs. Support was available for GTK-1 and GTK-2, Windows native WIN32, and QT. Currently, support is provided for GTK-2, Qt5, and Qt6.

H.1 HTML Viewer and Help Portal: `mozy`

Mozy is a multi-purpose HTML viewer derived from the help system used in the *XicTools* products from Whiteley Research Inc. These products are described on the Whiteley Research web site at `wrcad.com`. See 6.1 for a description of the *Xic* help system, and 6.1.2 for a description of the window controls.

There are a few command line options recognized. *Mozy* will take the first argument that is not an option as a topic to view. Recognized options are:

`--xic`
: This will cause *Mozy* to define the `Xic` flag in help text, i.e., help text enclosed in "`!!IFDEF Xic`" blocks will be read. The *Xic* help path will also be included in the default path. Thus, the text presented should match that as seen from running help within the *Xic* program. This option should be given if *Mozy* is being used to read the *Xic* help database.

`--wrs` or `--wrspice`
: Either of these will cause `Mozy` to define the `WRspice` flag in help text, i.e., help text enclosed in "`!!IFDEF WRspice`" blocks will be read. The *WRspice* help path will also be included in the default path. Thus, the text presented should match that as seen from running help within the *WRspice* program. This option should be given if *Mozy* is being used to read the *WRspice* help database.

The GTK2 graphical interface accepts the following options. These options are not processed by *Mozy*, but are intercepted by the graphics subsystem and affect the interface to the X-window system in Linux. The multiple forms are equivalent.

-d *dispname*
-display *dispname*
--display *dispname*
: This option specifies the name of the X display to use. The *dispname* is in the form

[*host*]:*server*[.*screen*]

The *host* is the host name of the physical display, *server* specifies the display server number, and *screen* specifies the screen number. Either or both of the *host* and *screen* elements to the display specification can be omitted. If *host* is omitted, the local display is assumed. If *screen* is omitted, screen 0 is assumed (and the period is unnecessary). The colon and (display) *server* are necessary in all cases. If no display is specified on the command line, the display is set to the value of the environment variable DISPLAY.

-name *string*
--name *string*
: This option provides an alternative name to the application, as known to the X window system. The application name is used by X to apply resource specifications.

--class *string*
: This option provides an alternative class name to the application, as known to the X window system. The application class name is used by X to apply resource specifications.

-synchronous
--sync
: This option indicates that requests to the X server should be sent synchronously, instead of asynchronously. Since the X system normally buffers requests to the server, errors do not necessarily get reported immediately after they occur. This option turns off the buffering so that the application can be debugged more easily. It should never be used with a working program.

--no-xshm *string*
: In releases running under the X-Window system (Unix/Linux), *Mozy* will use the MIT-SHM shared memory extension if the X server supports this extension, and the server is running on the local machine. This allows image data to be transferred to the X server via shared memory, which is faster than the normal X socket interface. Screen updates may be faster as a result.

: Giving the option --no-xshm on the command line will prevent use of this extension, if for some reason this is necessary.

Topics can also be entered by using the **Open** menu item in the **File** menu. A topic can be one of:

- The keyword associated with a help topic in the help database.
- A general URL referencing a page on the world-wide web. The URL must include the protocol specifier (“`http:`”) in the web address.
- The path to a viewable file on the local machine. A viewable file can be plain or HTML text, or an image.

Mozy displays level 3.2 HTML, and does not understand style sheets and consequently does a poor job displaying most current web sites. It works fine for basic HTML as likely found in help text, and in HTML email.

One application for *Mozy* is as an accessory to allow display of HTML messages from an email client such as `mutt` which does not have that capability. If, from `mutt`, HTML content is piped to *Mozy*, the viewer will appear displaying the content. Once visible, the operation can be repeated and the viewer will display the new content.

Mozy contains some unique features, provided in the menus. One such feature is the optional FIFO created in the user’s home directory. Text written to this “file” will be parsed and displayed. Another example is the **Log Transactions** button, which will cause the actual transmissions to and from the server to be duplicated to the standard output. This can be useful for debugging purposes. The **Bad HTML Warnings** button will issue warnings about imperfections in the HTML as it is parsed.

Mozy maintains a cache of pages and images, which is located in the subdirectory “`.wr_cache`” in the user’s home directory. If you see a really nifty web page, and you want to see the source, simply look at the `.wr_cache/directory` file. This will provide a listing of all of the components of the page, which are conveniently located in the same directory. The cache contents can also be viewed as a pop-up list from the **Show Cache** button in the **Options** menu. Clicking on an entry in the list will show that entry. Thus, you can revisit pages even when off-line.

Many of the features and capabilities of *Mozy* can be configured with a `.mozyrc` file placed in the user’s home directory. This is accomplished by pressing the **Save Config** button in the **Options** menu. Once this file is installed, it will be updated when viewer windows are closed, retaining the last settings.

H.1.1 *Mozy* Configuration

When *Mozy* starts, it will build an internal table of the topics found in the `.hlp` files that are found in the search path. The search path is a colon-separated list of directory paths, and if not given it will default to the single directory that corresponds to the default installation location for help files. This is `/usr/local/xictools/mozy/help`.

The search path is specified to *Mozy* through the `MOZY_HELP_PATH` environment variable. For example, suppose that the user has created some new `.hlp` files, and wishes to make them accessible to *Mozy*, while keeping access to the supplied files that are installed in the usual place. The user has placed the files, along with any needed image files, in `/home/joe/helpfiles`. If using `bash` or another Bourne-type shell, the command to set the environment variable would be

```
export MOZY_HELP_PATH=/home/joe/helpfiles:/usr/local/xictools/mozy/help
```

If using the C-shell, the corresponding command would be

```
setenv MOZY_HELP_PATH /home/joe/helpfiles:/usr/local/xictools/mozy/help
```

With this variable set in the environment, *Mozy* will be able to locate and display the user's topics.

When *Mozy* starts without an argument, a default topic is shown. The user may prefer that another topic be shown instead. This can be specified with the `MOZY_DEF_TOPIC` environment variable. This variable can be set to any keyword, URL, or local file path that could be given to *Mozy* through the **Open** command in the **File** menu. When the variable is set, the indicated page will be displayed when *Mozy* first appears.

The environment variables can be set in the user's shell startup script to make the definitions "permanent".

H.2 File Transfer Utility: `httpget`

The `httpget` program is a stand-alone utility for copying files from a remote system using the HTTP or FTP protocols. This is similar to the `httpget` program supplied with the XmHTML widget by Ripley Software Development with a few additions:

1. Support for FTP file retrieval.
2. Optional graphical window for status reporting.
3. Support for POST queries.
4. Support for HTTP basic authentication.
5. Support for transaction logging.
6. Conversion to C++ and (hopefully) useful classes.

The workings have been packaged into a library, `libhttpget`, which can be incorporated into other programs to provide in-process support for httpget-type functionality.

The `httpget` program is a command-line utility for retrieving files and posting queries to a remote HTTP or FTP server. It can be used, for example, from within a shell script to automate a software update. The program is invoked with

`httpget` [*options*] *url*

where the *options* are listed below, and the *url* is a standard-syntax universal resource locator, i.e., a web address of a file, in a form like

[`http://`]*server*/[*document*]

or

`ftp://`*host*/*file*

The *url* should contain the `http://` or `ftp://` prefix to indicate HTTP or FTP protocol, respectively. If no protocol is given, HTTP is assumed. The options are:

`-c` *file*
: Name of a cookie file. If not given no cookies are sent. Cookies will be stored in and dispatched from this file, during the transaction, if given.

`-d`
: Enable HTTP debug mode, by enabling printing of extra status messages during the transaction.

`-e`
: Don't reissue the request for HTTP location change error. Normal behavior if a 302 (location changed) response is received is to reissue the request to the new location. This option prevents this.

`-fp` | `-fh`
: Output format for errors: plain or HTML. Error messages are in plain text by default, but can be HTML formatted if `-fh` is given.

`-g`[x:y]
: Use a graphical window. If the graphics support has been included in the build, this option pops up a window which provides status indication and a **Cancel** button. Optionally colon-separated x/y root window coordinates can immediately follow, in which case the upper-left corner of the pop-up will be at that location, if allowed by the window manager.

`-h`
: Show help. The program lists these options and exits.

`-i`
: Only get HTTP document info (HEAD). The normal behavior is to retrieve the entire document. This option obtains document parameters only.

`-l` *file*
: Log bytes sent and received in *file*. The log file will contain a listing of the data transmitted and received.

`-n`
Don't print download status indication. Normal behavior is to print the number of bytes received. This option suppresses this.

`-o` *file*
Name of destination file on the local machine. If not given the standard output is used for HTTP and the host file name is used for FTP.

`-p` *proxy_url*
If given, the proxy will relay all transactions. The *proxy_url* must begin with an `http:` protocol specifier (`https` is not supported), and should have the port number appended following a colon, unless the default port 80 is used.

`-q` *file*
Query file for POST. The file is uploaded to the server.

The following two options set the time to wait for transmission. If contact is not achieved in the timeout interval, `httpget` will try again, up to the retry count.

`-r` *num*
Retry count, default is 0.

`-t` *num*
Timeout in seconds, default is 5.

`-s`
Save HTTP error to output on failure. The normal behavior is to emit errors to the standard error channel. With this option, errors are directed to the output channel (to the file if the `-o` option is given).

`-x`
Use HTTP error code as program exit value.

H.3 The *FastCap* Post-Processor: `fcpp`

Usage: `fcpp` *fastcap_outfile*

This program processes the output from the *FastCap* family of capacitance extraction programs, and prints the self and mutual capacitance values. It works with any known derivative of the original MIT *FastCap*, and with the *FasterCap* program from `FastFieldSolvers.com`.

H.4 Help to HTML Conversion Utility: `hlp2html`

Usage: `hlp2html` *path* [*path...*]

This program will convert a help database (`.hlp`) file to a collection of pure HTML files that can be viewed with a standard web browser. The program takes as argements paths to `.hlp` files, or paths to directories containing `.hlp` files, and converts the help text to ordinary HTML files in the current directory. The resulting set of files can be used to read the help text using an ordinary web browser.

WARNING: This program is ancient and rather obsolete, and does not handle the newer help database features.

H.5 Web Server Bridge to Help Database: `hlpsrv`

This is a server for the help database, which, through use of a simple cgi script, allows the help system to be accessed through a web server, such as Apache. Thus, the help database can be exported over the internet, as seen on the Whiteley Research web site `wrcad.com`.

There are three environment variables that must be set:

`HLPSRV_PATH`
The full real path to the help database files.

`HLPSRV_CGIPATH`
The server path to a cgi script, with argument prefix. This is prepended to the anchor text for all help keyword anchors. For example, the `.hlp` file contains

```
<a href="keyword">
```

and we have

```
HLPSRV_CGIPATH = "/cgi-bin/hlpsrv.cgi?h="
```

Then, the tag would be converted to

```
<a href="/cgi-bin/hlpsrv.cgi?h=keyword">
```

`HLPSRV_IMPATH`
The server path to images called from help database files. This is prepended to the path which follows `src=` in `img` tags. Images should be linked or moved to this location.

The invocation arguments are database keywords. The program dumps the `Content-type` header and topic text to the standard output.

Below is an example Apache cgi script, which makes available the *Xic* help database. This would be installed in the server's `cgi-bin` as (e.g.) `xichelp.cgi`.

```
#!  /bin/sh

IFS='='
set $QUERY_STRING

export HLPSRV_PATH="/usr/local/xictools/xic/help"
export HLPSRV_CGIPATH="/cgi-bin/xichelp.cgi?h="
export HLPSRV_IMPATH="/help-images/"

/usr/local/xictools/bin/hlpsrv $2
```

In web pages, the help system can then be accessed using code like

```
<a href="/cgi-bin/xichelp.cgi?h=xic">
Click here</a> to enter the on-line <b><i>Xic</i></b> help system.<br>
```

The word that follows `?h` is a help database keyword.

H.6 List File Pack/Unpack Utilities: `lstpack`, `lstunpack`

Utilities are provided to convert between different formats of list files, which are input files to the *FastCap* family of capacitance extraction programs. The capacitance extraction interface (see 16.17.1) generates this file format.

The original list file format, specified by MIT for the original *FastCap* program, actually used multiple files to describe the geometry. The list file references the other files.

The Whiteley Research revision of *FastCap* and the *FasterCap* program from `FastFieldSolvers.com` can make use of a "unified" list file format, where the geometry "files" are actually tacked onto the end of the list file itself, so that all input is provided in a single file. This can be much more convenient in cases where the input would otherwise require a large collection of files.

Whiteley Research provides two simple utilities to convert between formats.

`lstpack` *listfile*
: The *listfile* is an old-style (unpacked) list file, which references geometry files found in the same directory as the *listfile*. The utility will create, in the current directory, an equivalent unified list file. The base name of the file will be that of the input file, but with "`_p`" appended. The new file will have a "`.lst`" extension.

`lstunpack` *listfile*
: The *listfile* is a new-style (packed, or unified) list file. The utility will create, in the current directory, a new old-style unpacked list file, and all of the referenced geometry files. It would usually be wise to run this in a previously clear directory. The new file will have the same base name as the input file, but with "`_unp`" appended, and will be given a "`.lst`" extension.

Index

This page intentionally left blank.