Xic Reference Manual

Whiteley Research Incorporated
456 Flora Vista Avenue
Sunnyvale, CA 94086

Release 2.5.66
13 September 2006

© Whiteley Research Incorporated, 2006.

Xic is part of the *XicTools* software package for integrated circuit design. *Xic* was authored by S. R. Whiteley, with help from various public domain software libraries. Technical support for registered users of *Xic* and other *XicTools* software is available via electronic mail at `xic@wrcad.com`, and by telephone at (408) 735-8973.

Xic and subsidiary programs and utilities are offered as-is, and the suitability of these programs for any purpose or application must be established by the user as Whiteley Research, Inc. does not imply or guarantee such suitability.

This page intentionally left blank.

Contents

1 Introduction to *Xic* **1**

1.1 *Xic* Graphical Editor Overview . . . 1

1.2 Starting *Xic* . . . 3

1.2.1 Graphics Support and Requirements . . . 3

1.2.2 Command Line Options . . . 4

1.2.3 *Xic* Environment Variables . . . 7

Unix/Linux . . . 8

Microsoft Windows . . . 8

Xic Environment Variables . . . 8

1.2.4 *Xic* Search Paths . . . 11

1.2.5 Initialization Files . . . 14

1.2.6 Log Files and Error Reporting . . . 15

1.3 Menus and Operations . . . 17

1.3.1 Main Window Layout . . . 17

1.3.2 Main Drawing Window . . . 19

1.3.3 Prompt Line . . . 19

1.3.4 Layer Menu . . . 20

The S Button . . . 21

1.3.5 Status Display . . . 21

1.3.6 Coordinates Display . . . 22

1.3.7 Keypress Buffer . . . 22

1.3.8 Menu Selection and Accelerators . . . 22

1.3.9 Keyboard Input . . . 23

1.3.10 Mouse Buttons . . . 27

Basic Selection Operation . . . 27

Basic Move/Copy Operation . . . 29
Basic Stretch Operation . . . 29
Additional Notes . . . 30
Button 2 Operations . . . 30
Button 3 Operations . . . 31
Button 4, Mouse Wheel . . . 31
1.4 The Text Editor . . . 32
1.5 The **WR** Button: Email Client . . . 35

2 Using *Xic* 37

2.1 Physical Layout Editing . . . 38
2.2 Electrical Schematic Editing . . . 39
2.2.1 Placement of Devices and Subcircuits . . . 40
2.2.2 Semiconductor Devices . . . 41
2.2.3 Wiring Devices and Subcircuits . . . 43
2.2.4 Adding Properties to Devices . . . 44
2.2.5 Creating Subcircuits . . . 45
2.2.6 Node and Device Naming . . . 46
2.2.7 Generating Output and Running Simulations . . . 47
2.3 Cell Organization and Libraries . . . 48
2.4 Batch Mode . . . 49
2.5 Server Mode . . . 50
2.5.1 The Ascii Format . . . 53
2.5.2 The Binary Format . . . 54
2.5.3 Operation . . . 54
2.6 Template (Parameterized) Cells . . . 55
2.6.1 How Template Cells Work . . . 56
2.6.2 Creation of a Template Cell . . . 57
2.6.3 Adding an Instance of a Template Cell . . . 57
2.6.4 Changing the Parameters of an Instance . . . 58
2.6.5 Changing the Parameters of a Template-Derived Master . . . 59
2.7 Cadence^{TM} Compatibility . . . 59
2.7.1 Design Resource File . . . 60
2.7.2 Technology File . . . 60

3 The Help Menu: Obtain Program Documentation 65

3.1 The **Help** Button: Obtain Help . . . 65

3.1.1 The HTML Viewer . . . 67

3.1.2 The Help Database . . . 70

3.1.3 Help System Forms Processing . . . 70

3.1.4 Help System Initialization File . . . 70

3.2 The **About** Button: Program and Legal Info . . . 71

3.3 The **Release Notes** Button: View Release Notes . . . 71

4 The Side Menu: Geometry Creation 73

4.1 The **arc** Button: Create Arcs . . . 75

4.2 The **box** Button: Create Rectangles . . . 75

4.3 The **break** Button: Cut Objects . . . 76

4.4 The **deck** Button: Dump SPICE File . . . 77

4.5 The **desel** Button: Deselect Objects . . . 77

4.6 The **devs** Button: Device Menu . . . 77

4.6.1 Terminal Devices . . . 79

Ground Device . . . 80

Vcc Device . . . 80

4.6.2 SPICE Devices . . . 80

Resistor Device . . . 80

Capacitor Device . . . 80

Inductor Device . . . 81

Mutual Inductor . . . 81

Current Source . . . 81

Voltage Source . . . 82

Junction Diode . . . 82

Josephson Junction . . . 82

NPN Bipolar Transistor . . . 82

PNP Bipolar Transistor . . . 82

N-Channel Junction FET . . . 82

P-Channel Junction FET . . . 82

N-Channel MOSFET, 4 Nodes . . . 83

P-Channel MOSFET, 4 Nodes . . . 83

N-Channel MOSFET, 3 Nodes . . . 83
P-Channel MOSFET, 3 Nodes . . . 83
N-Channel MESFET . . . 84
P-Channel MESFET . . . 84
Transmission Line . . . 84
Transmission Line (LTRA compatibility) . . . 84
Uniform RC Line . . . 84
Voltage-Controlled Current Source . . . 84
Voltage-Controlled Voltage Source . . . 85
Current-Controlled Current Source . . . 85
Current-Controlled Voltage Source . . . 85
Voltage-Controlled Switch . . . 85
Current-Controlled Switch . . . 85
Example Opamp Macro . . . 85
4.7 The **donut** Button: Create Donut Object . . . 86
4.8 The **dots** Button: Show Connections . . . 86
4.9 The **erase** Button: Erase Geometry . . . 87
4.10 The **iplot** Button: Interactive Analysis Plotting . . . 87
4.11 The **label** Button: Create/Edit Labels . . . 88
4.11.1 Device Property Labels . . . 88
4.11.2 Spicetext Labels . . . 89
4.11.3 Hypertext . . . 89
4.11.4 "Long Text" Capability . . . 90
4.11.5 Script Labels . . . 91
4.11.6 Label Size Issues . . . 92
4.11.7 Label Font File . . . 93
4.12 The **layer** Button: Set Selection Mode . . . 93
4.13 The **logo** Button: Create Physical Text . . . 95
4.14 The **nodmp** Button: Enable SPICE Node Mapping . . . 97
4.14.1 Node Mapping . . . 97
4.15 The **plot** Button: Generate SPICE Plot . . . 98
4.16 The **polyg** Button: Create/Edit Polygons . . . 99
4.16.1 Polygon Vertex Editing . . . 99

4.16.2 Wire to Polygon Conversion . . . 100
4.17 The **put** Button: Extract From Yank Buffer . . . 100
4.18 The **round** Button: Create Disk Object . . . 100
4.19 The **run** Button: Run SPICE Analysis . . . 101
4.20 The **shapes** Button: Add Predefined Features . . . 103
4.21 The **sides** Button: Set Rounded Granularity . . . 104
4.22 The **spin** Button: Rotate Objects . . . 104
4.23 The **style** Button: Set/Change Wire Style . . . 104
4.24 The **subct** Button: Set Subcircuit Connections . . . 105
4.24.1 Virtual Terminals . . . 106
4.25 The **symbl** Button: Symbolic Representation . . . 106
4.26 The **terms** Button: Show Subcircuit Connections . . . 107
4.27 The **wire** Button: Create/Edit Wires . . . 107
4.27.1 Wire Vertex Editor . . . 108
4.28 The **xor** Button: XOR Objects . . . 108

5 The File Menu: *Xic* Input/Output 111

5.1 The **New** Button: Open Temporary Cell . . . 111
5.2 The **Open** Button: Pop Up File Selection Panel . . . 112
5.2.1 The File Selection Panel . . . 112
5.3 The **Save** Button: Save Cell . . . 114
5.4 The **Save As** Button: Save Cell, Renaming . . . 115
5.5 The **Print** Button: Print Control Panel . . . 116
5.5.1 Print Control Panel . . . 116
5.5.2 The **Format** Menu: Hardcopy File Formats . . . 118
5.5.3 `wr_lp` script . . . 120
5.6 The **Files List** Button: File Listing Panel . . . 121
5.7 The **Libraries List** Button: List Open Libraries . . . 122
5.8 The **Cells List** Button: Cell Listing Panel . . . 122
5.9 The **Show Tree** Button: Show Cell Hierarchy . . . 125
5.10 The **Quit** Button: Exit *Xic* . . . 125

6 The Edit Menu: Modify Geometry 127

6.1 Cell and Object Properties . . . 127

6.1.1 Electrical Mode Properties . . . 128
6.1.2 Physical Mode Pseudo-Properties . . . 129
6.2 The **Edit** Button: Edit Cells . . . 131
6.2.1 Input to the Edit Command . . . 132
6.2.2 Opening New Cells – Conflict Resolution . . . 133
6.2.3 Object Tests . . . 134
6.3 The **Current Transform** Button: Current Transform Panel . . . 135
6.4 The **Place** Button: Cell Placement Control Panel . . . 136
6.5 The **Push** Button: Push Editing Context . . . 138
6.6 The **Pop** Button: Pop Context . . . 138
6.7 The **Flatten** Button: Flatten Hierarchy . . . 139
6.8 The **Create Cell** Button: Create Symbol . . . 139
6.9 The **Properties** Button: Set Properties Panel . . . 139
6.9.1 The **Edit** Button: Edit Property . . . 140
6.9.2 The **Add** Button: Add New property . . . 141
6.9.3 The **Delete** Button: Delete Property . . . 142
6.10 The **Cell Properties** Button: Edit Cell properties . . . 142
6.11 The **Undo** Button: Undo Operation . . . 143
6.12 The **Redo** Button: Redo Last Undo . . . 143
6.13 The **Delete** Button: Delete Objects . . . 144
6.14 The **Erase Under** Button: Erase Under Objects . . . 144
6.15 The **Move** Button: Move Objects . . . 144
6.16 The **Copy** Button: Copy Objects . . . 144
6.17 The **Stretch** Button: Stretch Objects . . . 145
6.18 The **Chg Layer** Button: Change Layer . . . 146

7 The View Menu: Alter Presentation 147
7.1 The **View** Button: Select Cell View . . . 147
7.2 The **Physical** Button: Show Physical Mode . . . 148
7.3 The **Electrical** Button: Show Electrical Mode . . . 148
7.4 The **Expand** Button: Expand Subcells . . . 148
7.4.1 Peek Mode . . . 149
7.5 The **Zoom** Button: Zoom In/Out . . . 150
7.6 The **Viewport** Button: Create Subwindow . . . 150

7.7 The **Peek** Button: Show Layer Composition . . . 151
7.8 The **Cross Section** Button: Show Cross Section . . . 151
7.9 The **Rulers** Button: Create Rulers . . . 152
7.10 The **Info** Button: Display Information About Objects . . . 152
7.11 The **Bandwidth** Button: Show Rendering Time . . . 153
7.12 The **Allocation** Button: Show Memory Allocation . . . 153

8 The Attributes Menu: Set Display Attributes 155
8.1 The **Save Tech** Button: Update Technology File . . . 155
8.2 The **Key Map** Button: Assign Keys and Macros . . . 156
8.2.1 Key Mapping . . . 156
8.2.2 Key Mapping File . . . 156
8.2.3 Keyboard Macros . . . 157
8.2.4 Macro File Format . . . 158
8.3 The **Freeze Display** Button: Suppress Redisplay . . . 159
8.4 The **Constrain 45** Button: Constrain Angles . . . 159
8.5 The **Show Context** Button: Show Context in Subedit . . . 160
8.6 The **Show Properties** Button: Show Physical Properties . . . 160
8.7 The **Show Labels** Button: Display Labels . . . 160
8.8 The **Label True Orient** Button: Display Labels Transformed . . . 160
8.9 The **Show Cell Names** Button: Show Cell Names . . . 160
8.10 The **Cell Names True Orient** Button: Show Cell Names Transformed . . . 161
8.11 The **Show Origin Mark** Button: Mark Subcell Origins . . . 161
8.12 The **Tiny Boxes** Button: No Expansion of Tiny Subcells . . . 161
8.13 The **Set Font** Button: Set Window Fonts . . . 161
8.14 The **Set Grid** Button: Set Grid Parameters . . . 162
8.15 The **Set Color** Button: Set Colors Panel . . . 163
8.16 The **Set Fill** Button: Fill Pattern Edit Panel . . . 164
8.17 The **Edit Layers** Button: Edit Layer Table . . . 165

9 The Convert Menu: Data Format Conversion 167
9.1 Cell Name Aliasing . . . 168
9.2 Cell Name Alias File . . . 169
9.3 Layer Names . . . 170

9.4 Layer Filtering and Aliasing . . . 171
9.5 GDSII Layer Mapping . . . 172
9.6 The **Export Control** Button: **Conversion – Export** Panel . . . 173
9.6.1 CIF Dialect — Extension Styles . . . 175
CIF Cell Name Extensions . . . 175
CIF Layer Specification . . . 175
CIF Label Extensions . . . 176
9.6.2 Output Format . . . 176
9.7 The **Import Control** Button: **Conversion – Import** Panel . . . 178
9.8 The **Convert** Button: **Conversion** Panel . . . 180
9.8.1 Conversion: Notes and Limitations . . . 182
9.9 The **Read Into Current** Button: Read In Data from Another Cell . . . 184
9.10 The **Text Editor** Button: Edit Cell Text . . . 184
9.11 The **Edit Parameters** Button: Edit Conversion Parameters . . . 184
9.12 Skeleton Mode . . . 185

10 The DRC Menu: Design Rule Checking **189**
10.1 Layer Expressions . . . 190
10.2 Built-In Design Rules . . . 192
10.2.1 Global Rules . . . 193
10.2.2 Area Rules . . . 193
10.2.3 Edge Rules . . . 198
10.3 User-Defined Design Rules . . . 203
10.4 Assigning Design Rules . . . 210
10.5 The **Clear Errors** Button: Clear Error List . . . 212
10.6 The **Set Defaults** Button: Set Default Limits . . . 212
10.7 The **Set Flags** Button: Set Skip Flags . . . 213
10.8 The **Enable Interactive** Button: Set Interactive Checking . . . 213
10.9 The **No Pop Up Errors** Button: Suppress Error Report . . . 213
10.10 The **Check In Foreground** Button: Check Region or Cell . . . 213
10.11 The **Check In Background** Button: Initiate Check in Background . . . 214
10.12 The **Check In Region** Button: Check Objects . . . 214
10.13 The **Dump Error File** Button: Save Errors to File . . . 214
10.14 The **Show Errors** Button: Show Next Error . . . 214

10.15 The **Query Errors** Button: Print Error Text . . . 215
10.16 The **Edit Rules** Button: Rule Editor Panel . . . 215

11 The Extract Menu: Extraction and Verification 217
11.1 Extraction Setup and Configuration . . . 220
11.1.1 Layer Block Keywords . . . 221
11.1.2 Device Blocks . . . 225
11.1.3 Format Library File . . . 235
11.2 Extraction Methodology and Overview . . . 237
11.3 The **Select Groups** Button: Select Groups/Nodes . . . 238
11.4 The **Quick Paths** Button: Show Electrical Connectivity . . . 238
11.5 The **Show Paths** Button: Show Conducting Paths . . . 239
11.6 The **Show Groups** Button: Show Conductor Groups . . . 239
11.7 The **Show Nodes** Button: Show Node Numbers . . . 239
11.8 The **Show Terminals** Button: Show All Terminals . . . 240
11.9 The **Edit Terminals** Button: Place Connection Terminal . . . 240
11.10 The **Find Terminals** Button: Locate Terminal . . . 241
11.11 The **Source SPICE** Button: Update From SPICE File . . . 241
11.12 The **Source Physical** Button: Update Electrical From Physical . . . 242
11.13 The **Dump Phys Netlist** Button: Dump Physical Netlist . . . 243
11.14 The **Dump Elec Netlist** Button: Dump Electrical Netlist . . . 245
11.15 The **Dump LVS** Button: Test Layout vs. Schematic . . . 246
11.15.1 LVS Output File Format . . . 246
Conductor group and electrical node mapping . . . 246
Formal terminal group associations . . . 247
Physical device associations . . . 247
Physical subcircuit associations . . . 247
Checking per-group/node terminal references . . . 247
Summary . . . 248
11.16 The **Compare Devices** Button: Compare Extracted Parameters . . . 248
11.17 The **Compute Params** Button: Electrical Parameter Extraction . . . 248
11.18 The **Extract RLC** Button: Extract Parameters . . . 249
11.18.1 The FastCap/FastHenry Interface . . . 249
FastCap/FastHenry Interface Limitations . . . 250

Technology File Setup . . . 250
Geometry Specification . . . 253
Saving and Recalling Context . . . 254
Defining FastHenry Terminals . . . 254
FastHenry Partition Editor . . . 256
FastCap Partition Editor . . . 257
11.18.2 The RLC Extraction Panel . . . 258
11.19 The **Edit Extraction** Button: Set Parameters . . . 261

12 The User Menu: User Commands and *Xic* Scripts **263**
12.1 Script Menus: User-Defined Sub-Menus . . . 263
12.2 Introduction to *Xic* Scripts . . . 264
12.3 Encrypted Scripts . . . 265
12.4 The **Debug** Button: Enter Script Debugger . . . 266
12.5 The **Rehash** Button: Rebuild User menu . . . 268
12.6 Supplied Example Scripts . . . 269

13 The Scripting Language **271**
13.1 Error Reporting . . . 272
13.2 Data Types . . . 272
13.2.1 Scalars . . . 273
13.2.2 Strings . . . 273
13.2.3 Arrays . . . 275
Declaring and Defining Arrays . . . 275
Dynamic Resizing . . . 275
Pointers . . . 276
13.2.4 Handles . . . 277
13.2.5 Zoidlists . . . 278
13.2.6 Lexpers . . . 278
13.3 Math Operators . . . 279
13.3.1 Operator Overloading . . . 279
String Overloads . . . 280
Array Overloads . . . 280
Handle Overloads . . . 280

Zoidlist Overloads 281
13.4 Control Structures 281
13.4.1 `return` 281
13.4.2 `if, elif, else` 282
13.4.3 `repeat` 283
13.4.4 `while` 283
13.4.5 `dowhile` 283
13.4.6 `break` 283
13.4.7 `continue` 284
13.4.8 `goto, label` 284
13.5 "Preprocessor" Directives 285
13.6 Math Functions 285
13.7 User-Defined Functions 286
13.8 The `exec` Keyword — Immediate Execution 287
13.9 Static and Global Variables 288
13.10 Predefined Constants 290
13.11 HTML Forms and Scripts 290
13.11.1 Introduction to HTML Forms 290
13.11.2 Interfacing Forms to *Xic* Scripts 295
13.12 The Tcl/Tk Interface 296
13.13 Example Script 299

14 Keyboard '!' Commands **303**
14.1 Conversions 307
14.1.1 The **!assemble** Command: Merge Archives 307
14.1.2 The **!sced2xic** Command: Convert Old Format 310
14.1.3 The **!gzip** Command: Compress Files 310
14.1.4 The **!gunzip** Command: Uncompress Files 310
14.2 Current Directory 310
14.2.1 The **!cd** Command: Change Directory 310
14.2.2 The **!pwd** Command: Print Directory 310
14.3 Diagnostics 311
14.3.1 The **!logfiles** Command: Browse Log Files 311
14.3.2 The **!compare** Command: Compare Hierarchies 311

14.3.3 The **!empties** Command: Check for Empty Symbols . . . 312
14.3.4 The **!checkgrid** Command: Mark Off-Grid Vertices . . . 312
14.3.5 The **!checkover** Command: Report Subcell Overlap . . . 312
14.3.6 The **!dups** Command: Select Coincident Objects . . . 313
14.3.7 The **!check** Command: Database Consistency Check . . . 313
14.3.8 The **!regen** Command: Regenerate Labels . . . 313
14.3.9 The **!polycheck** Command: Check Polygon . . . 313
14.3.10 The **!polymanh** Command: Select Manhattan Polygon . . . 314
14.3.11 The **!polyfix** Command: Fix Polygon . . . 314
14.3.12 The **!polynum** Command: Number Vertices . . . 314
14.3.13 The **!bincnt** Command: Database Object Allocation . . . 314
14.3.14 The **!setflag** Command: Set Internal Cell Flags . . . 315
14.3.15 The **!summary** Command: Print Hierarchy Info . . . 315
14.3.16 The **!fileinfo** Command: Show File Statistics . . . 315
14.3.17 The **!time** Command: Show Elapsed Time . . . 315
14.3.18 The **!xdepth** Command: Show Transform Depth . . . 315
14.4 Design Rule Checking . . . 316
14.4.1 The **!showz** Command: Show DRC Test Areas . . . 316
14.4.2 The **!errs** Command: Regenerate DRC Error Markers . . . 316
14.4.3 The **!errlayer** Command: Create Error Polygons . . . 316
14.5 Extraction . . . 316
14.5.1 The **!source** Command: Source SPICE File . . . 316
14.5.2 The **!exset** Command: Extract Parameters and Set . . . 317
14.5.3 The **!addcells** Command: Add Missing Cells . . . 318
14.5.4 The **!find** Command: Find Devices . . . 318
14.5.5 The **!ptrms** Command: Default Terminal Locations . . . 319
14.5.6 The **!terms** Command: Regenerate Terminals . . . 319
14.5.7 The **!ushow** Command: Show Unassociated Elements . . . 319
14.5.8 The **!fx** Command: Control FastCap/FastHenry Interface . . . 320
14.5.9 The **!fxcell** Command: Create Cell From RLC Extraction Interface . . . 321
14.6 Fonts . . . 322
14.6.1 The **!dumpfont** Command: Dump Vector Font File . . . 322
14.6.2 The **!dumplogofont** Command: Dump Logo Font File . . . 322

14.6.3 The **!logofont** Command: Set Logo Font . . . 322
14.7 Graphics . . . 322
14.7.1 The **!setcolor** Command: Set Attribute Colors . . . 322
14.7.2 The **!display** Command: Export Rendering . . . 323
14.8 Grid . . . 323
14.8.1 The **!sg** Command: Save Grid in Register . . . 323
14.8.2 The **!rg** Command: Set Grid From Register . . . 323
14.9 Help . . . 323
14.9.1 The **!help** Command: Help Interface . . . 323
14.9.2 The **!helpfont** Command: Set Help Font . . . 324
14.9.3 The **!helpfixed** Command: Set Help Fixed Font . . . 324
14.9.4 The **!helpreset** Command: Clear Help Cache . . . 324
14.10 Layers . . . 325
14.10.1 The **!layer** Command: Generate Layers . . . 325
Advanced Features . . . 326
14.10.2 The **!ltab** Command: Modify Layer Table . . . 327
14.10.3 The **!lal** Command: Manipulate Layer Alias Table . . . 328
14.10.4 The **!ltsort** Command: Alphanumerically Sort Layer Table . . . 328
14.11 Libraries . . . 329
14.11.1 The **!mklib** Command: Create Library File . . . 329
14.12 Memory Management . . . 329
14.12.1 The **!clearall** Command: Clear All Memory . . . 329
14.12.2 The **!vmem** Command: Windows Virtual Memory Info . . . 329
14.12.3 The **!mmrecycle** Command: Set/Clear Recycle Mode . . . 329
14.12.4 The **!mmclear** Command: Clear Recycle Free Lists . . . 330
14.12.5 The **!mmstats** Command: Show Memory Manager Statistics . . . 330
14.13 Operations . . . 330
14.13.1 The **!mo** Command: Move Objects . . . 330
14.13.2 The **!co** Command: Copy Objects . . . 331
14.13.3 The **!ro** Command: Rotate Objects . . . 331
14.13.4 The **!rename** Command: Rename Cells . . . 331
14.13.5 The **!sa** Command: Save Cells . . . 331
14.13.6 The **!sqdump** Command: Save Selections as Native Cell . . . 332

14.13.7 The **!cont** Command: Read Contents of Native Cell . . . 332
14.13.8 The **!svq** Command: Save Selections in Register . . . 332
14.13.9 The **!rcq** Command: Recall Selections from Register . . . 332
14.14 Physical . . . 332
14.14.1 The **!box2poly** Command: Object Type Conversion . . . 332
14.14.2 The **!path2poly** Command: Outline to Polygon Conversion . . . 333
14.14.3 The **!poly2path** Command: Polygon to Outline Conversion . . . 333
14.14.4 The **!bloat** Command: Expand Objects . . . 333
14.14.5 The **!area** Command: Measure Layer Area . . . 335
14.14.6 The **!join** Command: Join Adjacent Objects . . . 335
14.14.7 The **!split** Command: Atomize Objects . . . 336
14.14.8 The **!manh** Command: Convert to Manhattan Polygons . . . 336
14.14.9 The **!polyrev** Command: Reverse Polygon Winding . . . 337
14.14.10 The **!noacute** Command: Eliminate Acute Angles . . . 337
14.14.11 The **!togrid** Command: Move To Grid . . . 337
14.14.12 The **!tospot** Command: Modify for Spot Size . . . 337
14.14.13 The **!bb** Command: Print Bounding Box . . . 338
14.14.14 The **!origin** Command: Move Cell Origin . . . 338
14.14.15 The **!import** Command: Import Cell Data . . . 339
14.15 Rulers . . . 339
14.15.1 The **!dr** Command: Delete Rulers . . . 339
14.16 Scripts . . . 340
14.16.1 The **!script** Command: Add Script . . . 340
14.16.2 The **!exec** Command: Execute a Script . . . 340
14.16.3 The **!lisp** Command: Execute Lisp Script . . . 340
14.16.4 The **!tk** Command: Execute Tcl/Tk Script . . . 342
14.16.5 The **!listfuncs** Command: List Saved Functions . . . 343
14.16.6 The **!rehash** Command: Rebuild User Menu . . . 343
14.16.7 The **!rmfunc** Command: Remove Saved Function . . . 343
14.17 Selections . . . 343
14.17.1 The **!select** Command: Select Objects . . . 343
14.17.2 The **!desel** Command: Deselect Objects . . . 345
14.17.3 The **!zs** Command: Zoom to Selected Objects . . . 346

14.18 Symbol Table . . . 346
14.18.1 The **!symtab** Command: Set Symbol Table . . . 346
14.19 Variables . . . 346
14.19.1 The **!set** Command: Set Variables . . . 346
14.19.2 The **!unset** Command: Unset Variables . . . 347
14.20 Window Cells . . . 347
14.20.1 The **!cc** Command: Load Cell into Window . . . 347
14.20.2 The **!sw** Command: Swap Cells in Main and Subwindow . . . 348

A File Formats **349**
A.1 Technology File . . . 349
A.1.1 Technology File Comments . . . 350
A.1.2 Technology File Macros . . . 351
A.1.3 Technology File Global Variables . . . 353
A.1.4 Technology File Path Definitions . . . 353
A.1.5 Technology File Scripts . . . 354
A.1.6 Technology File Layer Blocks . . . 354
A.1.7 Technology File Attributes . . . 361
A.1.8 Hardcopy Driver Parameters . . . 370
A.1.9 Example Technology File . . . 372
A.2 Resource File . . . 393
A.3 Symbol File Formats . . . 393
A.3.1 GDSII Format and Extensions . . . 394
Physical Mode Cell Properties . . . 395
A.3.2 The CIF File Format . . . 395
A.3.3 CIF Format Extensions . . . 398
A.3.4 Native Symbol File Format . . . 400
A.3.5 Computer Graphics Exchange (CGX) Format . . . 401
CGX Format Identifier . . . 401
CGX Data Types . . . 401
CGX Data Records . . . 401
A.4 Library Files . . . 406
A.4.1 Example Library File . . . 407
A.5 Device Library File . . . 407

A.5.1 Editing Devices . . . 411
A.6 Model Library Files . . . 414
A.7 Help Database Files . . . 415
A.7.1 Anchor Text . . . 418

B Property Specifications **421**
B.1 Global Properties . . . 421
B.2 Special Properties . . . 422
B.3 Physical Mode Properties . . . 422
B.4 Device Properties Set By *Xic* . . . 423
B.4.1 Node Property: Property number 10 . . . 424
B.4.2 Name Property: Property Number 11 . . . 425
B.4.3 Labloc Property: Property Number 12 . . . 425
B.4.4 Old Mutual Property: Property Number 13 . . . 426
B.4.5 Mutual Property: Property Number 14 . . . 426
B.4.6 Branch Property: Property Number 15 . . . 426
B.4.7 Labrf Property: Property Number 16 . . . 428
B.4.8 Mutlrf Property: Property Number 17 . . . 428
B.4.9 Symbolic Property: Property Number 18 . . . 428
B.4.10 Nodemap Property: Property Number 19 . . . 428
B.5 Device Properties Set By User . . . 429
B.5.1 Model Property: Property Number 1 . . . 429
B.5.2 Value Property: Property Number 2 . . . 429
B.5.3 Parameter Property: Property Number 3 . . . 430
B.5.4 Other Property: Property Number 4 . . . 430
B.5.5 NoPhys Property: Property Number 5 . . . 430
B.6 Special Escapes . . . 431

C *Xic* Variables **433**
C.1 Special Constructs . . . 437
C.2 Paths and Directories . . . 438
C.3 General Visual . . . 439
C.4 Keyboard '!' Commands . . . 440
C.5 Selections . . . 442

C.6 Side Menu Commands . . . 442
C.7 SPICE Interface . . . 444
C.8 File Menu — Printing . . . 445
C.9 Edit Menu Commands . . . 445
C.10 View Menu Commands . . . 447
C.11 Attribute Menu Commands . . . 447
C.12 Convert Menu — General . . . 448
C.13 Convert Menu — Input and ASCII Output . . . 448
C.14 Convert Menu — Output . . . 452
C.15 Extraction Menu Commands . . . 457
C.16 FastCap/FastHenry Interface . . . 461
C.17 Help System . . . 463

D Interface Functions 465

D.1 Control Functions . . . 481
D.1.1 Current Cell . . . 481
D.1.2 Database . . . 483
D.1.3 Display . . . 484
D.1.4 Exit . . . 486
D.1.5 Annotation . . . 486
D.1.6 Ghost Rendering . . . 487
D.1.7 Graphics . . . 489
D.1.8 Hard Copy . . . 492
D.1.9 Libraries . . . 496
D.1.10 Mode . . . 496
D.1.11 Prompt Line . . . 497
D.1.12 Scripts . . . 497
D.1.13 Technology File . . . 498
D.1.14 Variables . . . 498
D.1.15 *Xic* Version . . . 499
D.2 Utility Functions . . . 499
D.2.1 Arrays . . . 499
D.2.2 Bitwise Logic . . . 500
D.2.3 Cell Name Enumeration . . . 500

D.2.4 Error Reporting . . . 501
D.2.5 Generic Handle Functions . . . 502
D.2.6 Memory Management . . . 504
D.2.7 Miscellaneous . . . 504
D.2.8 Path Manipulation and Query . . . 504
D.2.9 Regular Expressions . . . 505
D.2.10 String List Handles . . . 505
D.2.11 String Manipulation and Conversion . . . 507
D.2.12 String and Array Size . . . 509
D.3 Operating System and Input/Output Functions . . . 509
D.3.1 Archive File Context . . . 509
D.3.2 Conversion . . . 517
D.3.3 Conversion — Export . . . 519
D.3.4 Conversion — Import . . . 521
D.3.5 Skeleton Mode . . . 521
D.3.6 Assembly Stream . . . 524
D.3.7 Current Directory . . . 526
D.3.8 Date and Time . . . 526
D.3.9 File System Interface . . . 527
D.3.10 Menu Buttons . . . 528
D.3.11 Mouse Input . . . 530
D.3.12 System Command Interface . . . 530
D.3.13 Text Input . . . 531
D.3.14 Text Output . . . 531
D.4 Geometry Creation and Management Functions . . . 533
D.4.1 General . . . 533
D.4.2 Current Transform . . . 533
D.4.3 Grid . . . 535
D.4.4 Layers . . . 536
D.4.5 Layer Alias Table . . . 541
D.4.6 Selections . . . 541
D.4.7 Object Management by Handles . . . 542
D.4.8 Pseudo-Flat Generator Functions . . . 553

D.4.9 Clipping Functions 554
D.4.10 Other Object Management Functions 557
D.4.11 Property Management by Handles 563
D.4.12 Other Property Management Functions 565
D.4.13 Design Rule Checking 566
D.4.14 Geometry Measurement 568
D.4.15 Zoidlists and Layer Expressions 569
D.4.16 Spatial Parameter Tables 574
D.4.17 Polymorphic Flat Database 575
D.5 Extraction Functions 577
D.5.1 Menu Commands 577
D.5.2 Terminals 580
D.5.3 Physical Conductor Groups 582
D.5.4 Physical Devices 583
D.5.5 Physical Subcircuits 584
D.5.6 Electrical Nodes 586
D.5.7 Electrical Devices 587

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. 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. Just below the main drawing

window is the prompt line. Below the prompt line is the layer menu, and to the left of the layer menu is the coordinate readout, and just below is the status area. Below the buttons in the side menu is the key press buffer area. The WR button in the upper left corner 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. A unique dual responsiveness to click and press/drag mouse input facilitates 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* through only a few keystrokes or mouse operations.

Xic includes a powerful script execution facility. A C-like scripting language is provided for implementing user-defined commands. *Xic* allows users to write custom commands which appear as buttons in a **User Menu**. These commands utilize a straightforward but powerful scripting language with a rich library of primitives for controlling the operation of *Xic*. A primary use of the user scripts is to create simple, menu based commands for creating certain geometrical objects or parameterized device structures, which are often needed in high frequency circuit design. 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 wab-based or turn-key third-party products or services, or in-house custom applications.

Hard copy support is available for a variety of printers and file formats, including PostScript (mono and color), Versatec (mono and color), HPGL, and HP laser. *Xic* is multi-lingual, with support for several archive file formats, and native input and output. Data input in a given format will remain in that format, unless explicitly converted.

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.

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, or an extension of GDSII, a binary format where the entire design is written to a single file, or an extension of CIF, an ASCII format where the entire design is written to a single file, 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 GDSII or CIF 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.

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.

The *XicTools* package has been developed primarily under BSD-4.4 Unix (FreeBSD, www.freebsd.org), which is the reference operating system. The tools have been ported to many other Unix-type operating systems, including Linux, Sun Solaris and SunOS 4.1.x, HPUX, and DEC Alpha-OSF. The tools are also now available for Microsoft Windows.

The Unix/Linux version of *Xic* uses the GTK+ user interface toolkit running on the X window system.

1.2 Starting *Xic*

1.2.1 Graphics Support and Requirements

Under Microsoft Windows, *Xic* uses the standard Microsoft interface, and operation should be familiar to Windows users. Although an attempt has been made to keep the user interface as close as possible to the Unix/Linux interface, there are subtle differences. Under Windows, *Xic* can support any number of colors, however if the user has a choice, using modes with more than 256 colors is recommended. The remainder of this section pertains to the Unix/Linux versions only.

Under Unix/Linux, *Xic* is designed to run under the X windowing system which is standard on these systems. The present version of *Xic* uses the GIMP Toolkit (GTK+), which is an open-source library of display objects. Previous versions of *Xic* used a customized version of the original Athena (Xt) widget set provided with the X-window system. The GTK+ interface provides a more powerful, more visually attractive, and more standard graphical user interface.

Support is provided for pseudo-color displays of eight planes or more, and for true-color displays of 16 planes or more. In a pseudo-color display, each pixel value is an index into an array of color values (the colormap), whereas in a true-color display, a pixel value is decomposed into red, green, and blue components which set the color without any mapping (often this is referred to as a "fixed colormap"). Pseudo-color displays are often more efficient for certain graphics operations, in particular blinking, since screen colors may be changed immediately by changing the mapping. In a true-color display, this would require actually redrawing areas of the screen. Most slightly older and low-cost workstations use pseudo-color displays. Modern PC hardware, on the other hand, provide both true-color and pseudo-color capability. The "16 bit" and "32 bit" modes are true-color, and the "8 bit" or "256 color" mode is pseudo-color. Typically, true-color is recommended if the user has a choice, as this eliminates the possibility of exhausting the colormap.

Xic will run on some other color systems, even gray-scale, though there may be shortcomings. In pseudo-color modes under X the display colormap is shared among all applications, thus if other applications have allocated too much of the colormap before *Xic* is started, *Xic* will create its own private colormap. This colormap will be installed when the keyboard is able to send characters to an *Xic* window, at which time windows of other applications may be shown in false colors. When the keyboard is not directing characters to an *Xic* window, *Xic* will be shown in false colors. Startup messages will report actions with respect to the colormap. This does not happen in true-color modes, as applications do not reserve pieces of the colormap in these modes.

In pseudo-color modes, the datum at the pixel address is used as an index into a colormap array, which contains red, green, and blue components for each entry. The size of this colormap determines the number of colors simultaneously available on-screen, which is usually 256. The strength of this approach is that ghosting and highlighting can be very efficient. Furthermore, on-screen colors can be dynamically adjusted by changing the colormap entries.

Graphics systems that use 16 bits or more per pixel generally contain no colormap. Instead, the datum is split into three components, the binary values of which control the red, green, and blue intensity. For example, the 16 bits are typically allocated into five bits for red and blue, and six bits for green. Thus, for each pixel, there are $32 \times 64 \times 32$ combinations of the red, green, and blue values. This is important when rendering images, graphics with shading, and the like.

The disadvantage of 256 colors is that this number is rapidly consumed by applications. What the world needs is 16-plane pseudo-color displays, but this would require about 200KB of high-speed static RAM for the colormap, which might be expensive. Under X, this problem is addressed with virtual colormaps. If an application can not reserve sufficient colormap space, the option exists for the application to create its own colormap, which will be loaded when the application has the keyboard focus. This was described above for *Xic*. The problem is that other windows will be shown in false colors, as will be the application when the focus is outside of the application. The color switching can be annoying.

Xic should be compatible with any window manager program. Some window managers, twm being an example, have a mode where windows are not automatically made visible when created, but instead must be located on-screen by the user clicking. The window outline is ghost-drawn, attached to the pointer. Although *Xic* can be used in this mode, it is not efficient. The window manager should be set to automatically realize newly created windows. See the documentation for the window manager for information on how to change the defaults, if it becomes necessary to make this change. In twm, giving the RandomPlacement keyword in the startup file solves this problem.

Most window managers provide a choice between "click to focus" and "focus follows mouse" modes for specifying which window will receive keyboard input. In the former, one must click the border of a window for that window to receive keyboard input, and that window will retain the "focus" until another window is selected. In the latter case, keyboard input is always directed to the window containing the mouse pointer. Which system to use is a matter of personal preference, though the "focus follows mouse" mode is usually deemed more efficient.

Xic expects button events from buttons 1–3, and the same buttons with the **Shift** and **Ctrl** keys pressed. The three buttons (a three button mouse is recommended) are normally numbered from the left, with the mouse pointing upward. The **Shift**-button 1 and **Ctrl**-button 1 events are most important, **Shift**-button 2 is used only peripherally (in the fill pattern editor), and **Shift**-button 3, and **Ctrl**-buttons 2,3 are not presently used but may be used in future versions of *Xic*. The window manager should be set to not intercept these events, particularly those for button 1. If issuing **Shift**-button 1 brings up a menu of window manager options, for example, then the window manager mappings must be modified. Presently, it is not generally possible to modify the *Xic* button mappings at the user level.

1.2.2 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 **Edit** command.

```
xic [-T[extension]] [-E] [-Ggeometry_spec] [-Hdirectory_path] [-Kpassword] [-Lserverhost]
  [-S[port] | -Bbatch_opt] [-C | -C1] [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.

-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.

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

-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.

-L*serverhost*
: This supplies the host name of the machine running the license server. Note that there is no space after `-L`. If given, this will override the server host supplied by other means.

Below is the logic hierarchy for setting the license server host, each method will override those listed lower. See the documentation for the `xtlserv` (license server) program for more information.

-L*serverhost*
XTLSERVER in environment
`license.host` file
`xtlserver` in `/etc/hosts`
name of local machine

-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 2.5. The option can be immediately followed (no space) by a port number to use for connections.

-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*

Batch mode will be described in 2.4.

The -C and -C1 options apply only to "pseudo-color" displays. These are displays with "8-bits" or "256 colors", found frequently on Sun 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
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
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.

If colormap space is short, *Xic* will allocate single plane cells. *Xic* will also use single plane cells if the -C1 option is given on the command line. This of course uses 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.

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 and 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 and 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*
If this option is given, shared memory will not be used between the application and an X server on the same machine, if shared memory is available. There is no reason to use this option except for program debugging purposes.

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 **Edit** command.

1.2.3 *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*.

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.

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.

Xic Environment Variables

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

XT_AUTH_MODE

Normally, Unix/Linux versions of *Xic* and *WRspice* use authorization provided by an external license server, possibly hosted on a different machine. On the other hand, the Windows versions and *XicII* use built-in authorization. Both the external license server and the programs not using the license server make use of a file named "`LICENSE`" provided by Whiteley Research, Inc., which provides authorization to run on the host computer.

It is possible to run *Xic* and *WRspice* without a license server, and to run *XicII* from a license server. The status is set with the environment variable XT_AUTH_MODE. This variable has meaning if set to one of the keywords "`Server`" or "`Local`"

If set to "`Server`", *XicII* will validate through a license server the same way as *Xic* and *WRspice* normally do. If set to "`Local`", *Xic* and *WRspice* will be self-validating the way *XicII* normally is.

Xic and/or *WRspice* users on a single licensed workstation may prefer to set the environment variable in their shell startup file and not use the external license server.

When the programs look for the LICENSE file in "`Local`" mode, if the file is not found in the startup or license directories, the programs will look in the home and current directories, in that order, unless XT_LICENSE_PATH is also set.

XT_LICENSE_PATH

When using local validation (i.e., not using the license server) XT_LICENSE_PATH can be set to the full path to the license file. Only this file will be used – the regular search is suppressed.

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.

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. This should not be needed under Windows, as the Registry provides the default paths.

XT_USE_GTK_THEMES

There is a potential portability issue with the GTK theme engine which is dynamically installed on some systems (Unix/Linux only). Unless the engine supplied on the user's system is compatible with the libraries used by *Xic*, dynamic runtime linking will fail, and in at least one case *Xic* will exit. To avoid this problem, *Xic* uses a default theme known to work (see `README` under `default_theme` in the startup directory). This can be disabled, i.e., the system themes used, by setting the environment variable XT_USE_GTK_THEMES This applies to *Xic* and *WRspice*.

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 to the right of the main window. The default to to place the menu at the left. This works with Unix/Linux only, the menu is always on the left under Windows.

XIC_START_DIR, HOME

Under Windows, if *Xic* finds itself in the `C:\` directory on startup, it will attempt to change the working directory. This is the default when starting from the Windows Start Menu, unless the menu property is changed. If a variable XIC_START_DIR is set, *Xic* will assume the value is a directory path, and change to that directory. If not set, *Xic* will look for a variable named HOME and if found will assume the value is a directory path and change to that directory. If neither is found, or the directory change fails, *Xic* will remain in the `C:\` directory.

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.

XEDITOR_FONT

The text editor windows will use a font defined in the XEDITOR_FONT environment variable. 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 10`" or "`(10)Lucida Console`".

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/lib/xictools/xic/docs`, or, if XT_PREFIX is set, its value will replace `/usr/local`.

XT_REMOTE_TEST, XTNETDEBUG

There are two environment variables used for debugging the network interfaces. If XT_REMOTE_TEST is defined, *Xic* will link to *WRspice* on the local machine through the `wrspiced` daemon. Ordinarily, *Xic* will fork a *WRspice* process directly on the local machine. 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.

XICNOGDB
: If *Xic* should crash, the normal action is to use the Unix `gdb` program to obtain a stack backtrace, which is emailed to Whiteley Research, along with the last few lines of the `xic_run.log` file, for analysis. If the variable XICNOGDB is defined, this will not happen. A similar process occurs under Windows, which can likewise be suppressed with the XICNOGDB variable.

XICNOMAIL
: If the variable XICNOMAIL is set, no mail will be sent during a crash. Under Windows, the file that would have been sent is located in the current directory, and is named "`xic.stackdump`". Under Unix/Linux, a file named "`gdbout`" is produced if this variable is *not* set, if the `gdb` debugger is present on the system, and XICNOGDB is not set. If XICNOMAIL is set, and XICNOGDB is not set, then the dead *Xic* process will be placed under the control of the debugger.

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.

XT_SYSTEM_MALLOC
: The Unix/Linux *Xic* releases have built-in memory management. The built-in memory manager allows *Xic* to use all available system memory, which is not generally true with the standard memory manager supplied with the operating system. The standard memory manager often allows *Xic* to use less than 1Gb of memory before an out-of-memory error will halt the program. The built-in memory manager will allow use of more than this, assuming that it is available on the system.

  If *Xic* is started with the XT_SYSTEM_MALLOC environment variable set, then *Xic* will use the standard memory manager provided by the operating system.

1.2.4 *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. 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 symbol path, the library path, the help path, and the script path. The symbol path is used to locate symbol files, which contain editable cells and archives. 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 symbol path is augmented with any path preceding a cell to open in the **Edit** command. By default, for native cells, the path is added to the beginning of the present symbol 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 **Edit** button, then the file maincell is opened for editing, and the directory `/usr/work` is added to the front of the symbol path. Once the symbol 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 cell to open in the **Edit** command can be altered with two variables, which can be set with the **!set** command. These are described in 6.2.

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:

Symbol Path
- variable: Path
- environment: XIC_SYM_PATH
- default: `( . )`

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

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

ScriptPath
- variable: ScriptPath
- environment: XIC_SCR_PATH
- default: `( /usr/local/lib/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 symbol data, whether *Xic*, GDSII, or some other format, are expected to be found in a directory along the symbol search path. The first file found matching the name requested is opened. Normally, it is desirable to include the current directory '.' in the symbol path, otherwise symbol 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.

1.2.5 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.

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.2).

Next, an "`.xicinit`" initialization script, if present, will be read and executed. The initialization script uses exactly the same format as other script files, as are normally found along the script search path. The scripting language is described in detail in Chapter 12. 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.

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.

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.

If a key mapping file is found in the library search path, that file is read to provide a mapping for certain keyboard keys which may not be mapped as expected. This file is generated from within *Xic* with the **Key Map** command, and is not normally edited by the user.

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).

Next, *Xic* will attempt to read a file with the base name "`.xicmacros`", 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 `./.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. This file holds 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.

Next, an initialization script, if present, will be read and executed. 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 files `./.xicstart.ext` and `$HOME/.xicstart.ext`, then `./.xicstart` and `$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.

The legal message is read from a file named "`xic_mesg`" found in the library search path. This message appears when the **About** button in the **Help Menu** is selected.

As needed, *Xic* will also read the `device.lib` file, search and map the device models and help files, and open the first command line file for editing. The `device.lib` 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.

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.

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 technology file contains all of the information *Xic* needs for physical layout, 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 default technology file, plus several other files needed, are placed in a system-wide location on installation, usually `/usr/local/lib/xictools/xic/startup`. 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. The default technology file provided with *Xic* is for generic MOSIS scalable CMOS.

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. The technology file format is described in the Appendix.

1.2.6 Log Files and Error Reporting

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 **!logfiles** command.

`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 10 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 XICNOMAIL 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.

`convert_rdcgx.log`
`convert_rdcif.log`
`convert_rdgds.log`
`convert_rdoas.log`
`convert_rdxic.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.

`convert_tocgx.log`
`convert_tocif.log`
`convert_togds.log`
`convert_tooas.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_frcgx.log`
`convert_frcif.log`
`convert_frgds.log`
`convert_froas.log`
: These files contain messages emitted when a non-native 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.

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:

- All cells in memory that have the modified flag set will be written to the current directory, with a suffix "`.panic`" added to the normal file name. 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 XICNOGDB 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 "`xic.stackdump`" under Windows, or "`gdbout`" under Unix/Linux. The file will be created in the current directory, but under Unix/Linux, only if the `gdb` debugger is present on the system.

1.3 Menus and Operations

1.3.1 Main Window Layout

Figure 1.1 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.

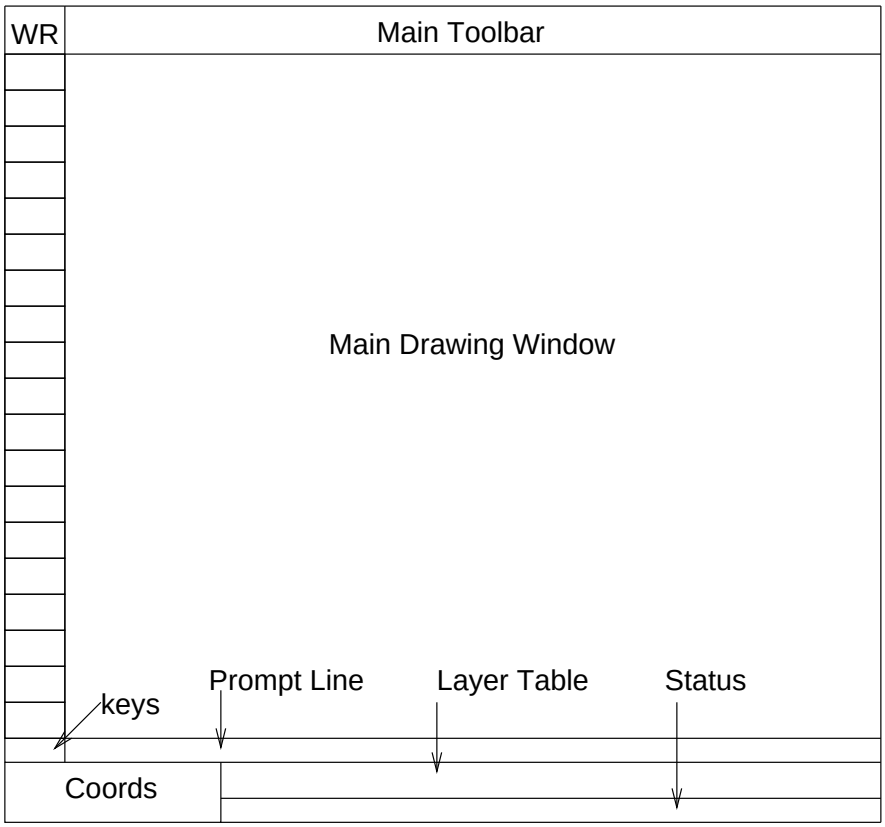
Xic has nine drop-down menus, arrayed in a menu bar which extends across the top of the main application window.

File Menu	Commands to open, save, and list files and cells. This menu also contains the printer interface.
Edit Menu	Commands which are used to modify the current design.
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.

In addition to the menu bar menus, *Xic* has a side menu of buttons, displayed along the left of the main window. These are generally used to create a specific type of feature, such as a rectangle. It is possible to start *Xic* so that the side menu is on the right, by setting the environment variable XIC_MENU_RIGHT.

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

Figure 1.1: Default *Xic* screen layout. It is also possible to place the side menu to the right of the main drawing window.



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 with the !help keyboard command. The argument to this command is "`xic:`" followed by the command name, for example "`!help xic:prpty`".

1.3.2 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.

Xic supports the xdnd and Motif 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 **Open** 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 drag 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.

1.3.3 Prompt Line

The prompt line is a dialog box just below the main drawing window, and above the layer menu. Messages and prompts are displayed in this area, as well as textual input to *Xic*. The prompt line participates in the drop protocol for files. Files dropped on the prompt line have the same effect as files dropped in the drawing windows. When in text editing mode, the full path to the file is inserted into the text line. Otherwise, the file will be opened for editing as *Xic* input. This means that when responding to a prompt to open a file, the **File Selection** pop-up from the **Open** 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.

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 **Edit** 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.

Xic contains a "hypertext" editing capability, which is active in electrical mode. This is necessary, for example, when setting device properties which reference other devices or nodes. The device names and node numbers are set by *Xic*, and change if the circuit is modified, 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 accesses 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 **Bsp** keys, the hypertext references are treated as single items.

In text edit mode, the arrow keys move the cursor back and forth, **Bsp** 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-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, which is the normal way to terminate input.

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 more fully in the description of the **label** command (4.11).

1.3.4 Layer Menu

The layer menu is arrayed beneath the main drawing window and the prompt line. If layers have been specified to *Xic*, they will be shown in this area. Each layer contains a sample area and name. If there are more layers than space available for display, clicking on the arrow buttons to the right of the layer menu will show a new line of layer entries.

The up arrow, with accelerator keyword "`ltup`", advances to the set of layers above those presently listed. The down arrow, with accelerator keyword "`ltdn`", scrolls in the opposite direction. Both operations wrap around the ends of the list of layers.

Immediately to the left of the the layer table is the **S button**. When this button is in the down position, *Xic* is placed in layer-specific mode for selections.

One of the layers will be the "current layer", which is assumed by many of the drawing commands. The current layer is highlighted in the layer table, and may be changed by clicking with button 1 on the entries in the table. The current layer is used for all created geometry, and for layer-specific selections. Clicking on the layer entries with button 2 toggles the layer visibility on and off, as shown by the presence or absence of the sample area. Button 3 toggles or enables layer blinking. In 256-color modes, the blinking status is turned on and off by clicking with button 3. In other modes, button 3 must be pressed and held over a layer entry, and only the main drawing window will display blinking. In either case, 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.

A small icon at the far right of the layer table indicates when layer-specific selection mode is active. Clicking on this icon with button 1 will toggle layer-specific mode on and off. Clicking on this icon with button 2 will toggle the visibility of all layers. Thus, if the user wants to view only one layer, a quick way to achieve this is to click on the icon with button 2 to turn off all layers, then click on the

layer(s) to show with button 2. Note that unless the graphics is set for 256-color mode, the screen has to be explicitly redrawn (**Ctrl-R** does this) to show the new view in physical mode. Holding **Shift** while clicking with button 2 will force a screen redraw.

The entries in the layer table can be shown in more compact form by setting the variable LayerTable, with the **!set** command, to either "`compact`" or "`tiny`". For example, one can type

```
!set LayerTable compact
```

to obtain the compact representation. Entering

```
!unset LayerTable
```

reverts to the default view.

The compact view uses smaller sample boxes and text. The tiny view uses still smaller sample boxes, and two rows.

There is no limit on the number of layers that can be defined in *Xic*.

Each layer has an optional long name in addition to the CIF style name shown in the layer menu. When specifying a layer to a command or function, either name can be used.

The S Button

Immediately to the left of the the layer table is the **S button**. When this button is in the down position, *Xic* is placed in layer-specific mode for selections.

In layer-specific mode, only objects on the current layer, or subcells containing objects on the current layer, can be selected and deselected with mouse operations. Otherwise, any visible object can be be selected or deselected. Instances are selected during a layer-specific point select if there is no other qualifying geometry where the user clicked, and only instances that contain objects on the current layer can be selected in layer-specific mode. The layer-specific mode affects many of the commands and operations in *Xic*, typically limiting the operation to objects on the current layer when active.

Layer-specific mode is indicated by a small box icon in the far right side of the layer table. Clicking on this icon with button 1 will also toggle layer-specific mode on and off.

Layer-specific mode can also be set from the **Layer Specific** button which appears in the **Selection Control Panel**, which is brought up by the **layer** button in the side menu.

1.3.5 Status Display

The status area is located below the layer menu. This area provides information about current program modes. It displays 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 "`Idle`", 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 physical grid origin is not 0,0 (set with the PhysGridOrigin variable), "`PhGridOffs`" will be displayed in colored text.

1.3.6 Coordinates Display

In the lower left corner of the *Xic* display 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.

1.3.7 Keypress Buffer

Below the buttons in the side menu is the key press buffer area. This area displays the last five keys in the keypress buffer. 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. In Windows, menu items do not have tooltips, instead, the tooltip text is displayed in the status line.

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.

While *Xic* is awaiting input in the prompt line, the word "INPUT" is shown in colored text in this window, and the prompt line background switches to a lighter color.

1.3.8 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 toolbar. If this is followed by a character underlined in one of the menu entries, that

function is invoked. For example, typing **Alt**-**Ep** (press and hold **Alt**, press **e**, release **Alt**, press **p**) engages the **Push** command in the **Edit Menu**.

2. If the menu entry has something in the second column, that is also an accelerator. For example, in the **File Menu**, the **Open** entry has "Ctrl-O" listed in the second column. This indicates that pressing **Ctrl**-**O** will invoke the **Open** 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 **Key Map** command in the **Attributes Menu**.

Finally, there are a few hard-coded keyboard commands, listed below, that invoke functions that are also available in the menus.

1.3.9 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 the word "INPUT" will be printed in the keys buffer area in colored text, and the prompt line background will appear in a lighter color. Otherwise, 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 **Bsp** (Backspace) key will delete the last character entered. There are other accelerators for most menu commands.

When entering text into a text input window or onto the prompt line, several keys provide editing capability. The cursor can be moved with the arrow keys. The **Delete** key will delete the character over the cursor, and the **Bsp** key (or **Ctrl**-**H**) will delete the character to the left of the cursor. **Ctrl**-**U** will delete the entire line. **Ctrl**-**A** will move the cursor to the beginning of the line, and **Ctrl**-**E** will move the cursor to the end of the line. **Esc** will terminate the command without saving input. **Enter** will end editing and send the line to *Xic*. The cursor can be at any position when **Enter** is pressed.

There are several other keys with special significance to *Xic*. Many of these keys should be memorized by the user, as there is no alternative way to invoke their function.

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 "`Idle`".

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 **Edit 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 **Edit Menu**.

Pressing the **Delete** key will delete any objects currently selected. Objects in a drawing can be selected with button 1 operations (see 1.3.10). This has the same effect as the **Delete** button in the **Edit Menu**. If the **Rulers** button in the **View Menu** is active, the **Delete** key will delete rulers and not other objects.

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** 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).

Holding **Ctrl** while pressing the arrow keys will cycle the current rotation setting, otherwise set with the **Current Transform** command in the **Edit Menu**. This affects moved and copied objects and new instances.

Holding both **Shift** and **Ctrl** while pressing the arrow keys will cycle through the previous views in the window containing the mouse pointer. 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.

While in a command that is prompting for text input, the arrow keys, pressed alone, will move the text cursor back and forth, and not pan the display. The **Shift** and **Ctrl** keys, when held while pressing the arrow keys, produce the same effects described above, so fine-panning is still available during text input.

The arrow keys may have special functions in individual commands, which override the behavior above. This is noted 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 14 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.1. 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 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.

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, the **Pause** key also sends an interrupt signal, and unlike **Ctrl-C** this is independent of which *Xic* window 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/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** prompts the user for new grid parameters. The grid can also be set with the **Set Grid** button in the **Attributes Menu**.

The user is prompted with a default command string containing three tokens: the grid spacing in microns, the snap number, and on/off. The snap number determines where the pointer can actually locate a coordinate. If the snap number is positive, then it represents the number of possible positions per grid spacing. If two, for example, the grid lines and midway between the grid lines are coordinates that can be accessed. If the snap number is negative, then the value represents the number of grid lines per snap line.

If the returned string is blank, the grid visibility is toggled. Entering "**space Enter**" is a quick way to toggle the grid. If the returned string contains one token, it is taken as a new grid spacing if a number, or sets the grid visibility on if "`y`" or "`on`", and off if "`n`" or "`off`". If there are two tokens, the first token is a new grid spacing, and the second token is either a snap number or an on/off directive if not a number.

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

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. An important example of their use will be presented in the next section. The sections describing the commands will present more examples.

1.3.10 Mouse Buttons

Xic is most efficiently used with a three-button mouse. 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. A mouse wheel will 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. If **Shift**, **Ctrl**, and **Alt** are all held while button 1 is pressed, a button 4 press is simulated. This performs no action, but updates the coordinate readout window. The following sections describe the normal operations.

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 **layer** button in the side menu.

In layer-specific mode (see 1.3.4), only objects on the current layer or subcells containing objects on the current layer can be selected or deselected. Otherwise, all objects and subcells are candidates for selection and deselection. Objects remain selected if the current layer is changed, and all selected objects will be moved, copied, or stretched. In layer-specific mode, the user must initiate the move, copy, or stretch on a selected object on the current layer.

Clicking on a single object will toggle the selection status of the object, if it is in scope according to the layer-specific mode. 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 layer-specific mode, objects not on the current layer are ignored, whether or not they are selected.

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, or all those on the current layer if in layer-specific mode. 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. All objects are affected if not in layer-specific mode, and objects on the current layer are affected otherwise. 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.

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.

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.

If the **Shift** key is down while the user completes the operation, by either releasing button 1 if dragging or clicking button 1 a second time, the objects will be copied to the new location. If **Shift** is not down when the operation is completed, the objects will be moved.

If the **Ctrl** 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. 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 will be placed on the new layer. Subcells are not affected. If in layer-specific mode, only objects whose layer was the original current layer will be changed to the new layer. If not in layer-specific mode, all new objects will be placed on the new layer, no matter what their original 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".

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 not available from the basic operation. Stretching operations are also available for polygons in the **polyg** command, and for wires in the **wire** command.

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.

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.

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 subwindow, 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 subwindow. The subwindow 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.

Many of the Pop-up windows can be canceled by double clicking with button 2 in the background, or on a label object in the pop-up. The cursor changes to the skull and crossbones after the first click, indicating cancellation on the next button 2 click. Moving the pointer off the widget resets the pending cancellation, returning the cursor to the default.

Button 3 Operations

Button 3 performs a zoom operation. Clicking in two locations defines a rectangle to zoom into. The window will then display the contents of this area (after compensating for aspect ratio). To zoom out, button 3 is held while the pointer is moved, then released. A dotted box is drawn defined by the press and release locations. A second button 3 press in the same window will cause the area defined by the first and second presses to be shrunk by the ratio of the diagonals of the rectangles defined by press 1, release 1 and press 1, press 2. To zoom out a lot, release 1 is much closer to press 1 than press 2. A second 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 pressing button 3 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 will cause the layers to blink. In 256-color modes, the blinking status is turned on and off by clicking with button 3. Clicking with button 3 on the small box icon at the far right of the layer menu will turn off blinking of all layers, in this case. In other modes, button 3 must be pressed and held over a layer entry, and only the main drawing window will display blinking. In either case, 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.

Button 4, Mouse Wheel

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.

Under Unix/Linux, the GTK+ user interface provides support for mouse wheels, by assuming that up and down clicks are mapped to button 4 and 5 presses. In order to use the mouse wheel, the X server must enable this function. For FreeBSD and Linux, this is enabled in the `XF86Config` file in `/usr/X11R6/lib/X11`, by adding the "ZAxisMapping" keyword to the "Pointer" section, as shown in the example below.

```
Section "Pointer"
```

```
  Protocol        "Auto"
  Device          "/dev/psm0"
  ZAxisMapping    4 5
EndSection
```

Any window that has scroll bars can be scrolled by moving the pointer *over a scroll bar* and turning the mouse wheel. The text windows and help viewer windows in *Xic* respond to the mouse wheel by scrolling when the pointer is in the text window, as well as over the scroll bar.

1.4 The Text Editor

Xic 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.

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 **Open** 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. 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.

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. When this pop-up is visible, and a file is dragged into the editor and dropped, the name of that file is entered into the dialog text area.

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 DOS
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 DOS two-byte termination. If this button is unset, the more modern and efficient Unix-style termination is used. Older Windows

programs such as Notepad require DOS 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 following commands are found in the **Edit** menu of the text editor.

Cut
Delete selected text to an internal buffer (the "clipboard").

Copy
Copy selected text to the clipboard.

Paste
Paste the contents of the clipboard into the document at the cursor location.

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.

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.

Unless a font is set with the **Font** button, text editor windows will use a font defined in the XEDITOR_FONT environment variable, if set. 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 10`" or "`(10)Lucida Console`".

Characters are entered into the document as typed, at the current cursor location. The keys with special bindings are listed below.

Unix/Linux Bindings

Ctrl-A	Move cursor to beginning of line
Ctrl-B	Move cursor backward one character
Ctrl-C	Copy selected text to clipboard
Ctrl-D	Delete next character
Ctrl-E	Move cursor to end of line
Ctrl-F	Move cursor forward one character
Ctrl-H	Delete previous character
Ctrl-N	Move cursor down one line
Ctrl-P	Move cursor up one line
Ctrl-U	Delete current line
Ctrl-V	Paste clipboard at cursor
Ctrl-W	Delete backward one word
Ctrl-X	Cut selection to clipboard
Alt-B	Move cursor backward one word
Alt-D	Delete word
Alt-F	Move cursor forward one word
Home	Move cursor to beginning of line
Ctrl-Home	Move cursor to top of document
End	Move cursor to end of line
Ctrl-End	Move cursor to end of document
PageUp	Move up one page
PageDown	Move down one page
Up	Move cursor up one line
Down	Move cursor down one 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 clicked-on 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 contents of the clipboard (or any selected text) into the document at the click location. Button 3 will also select text, but the selected text is not copied to the clipboard.

Microsoft Windows Bindings

Ctrl-A	Select all text
Ctrl-C	Copy selected text to the clipboard
Ctrl-V	Paste the clipboard contents at the cursor
Ctrl-X	Cut selected text to clipboard
Backspace	Delete selected text, or the character to the left of the cursor if no text is selected
Ctrl-Backspace	Delete the word to the left of the cursor
Delete	Delete selected text, or the character to the right of the cursor if no text is selected
Shift-Delete	Performs a "cut", i.e., copies to the clipboard before deleting
Ctrl-Delete	Delete the word to the right of the cursor
Insert	Toggle insert/overwrite mode for typed characters
Shift-Insert	Perform a "paste" from the clipboard
Home	Move cursor to the start of the line
Ctrl-Home	Move cursor to the top of the document
End	Move cursor to the end of the line
Ctrl-End	Move cursor to the end of the document
Page Up	Scroll up one page
Ctrl-Page Up	Move cursor to start of first visible line
Page Down	Scroll down one page
Ctrl-Page Down	Move cursor to end of last visible line

The arrow keys move the cursor, and with **Ctrl** pressed the left and right arrow keys move the cursor word-by-word. Holding the **Shift** key while moving the cursor with the mouse or with other keys will select the intervening text. The left mouse button is used to move the cursor by clicking, and to select text by dragging.

1.5 The WR Button: Email Client

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 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 menu 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. The mail system is known to work if Outlook Express is installed and configured as the "Simple MAPI mail client". To configure this, from Outlook Express select Tools\Options\General from the menu. Then, check the box next to "Make Outlook Express my default Simple MAPI client". Outlook Express is installed automatically with Internet Explorer.

Chapter 2

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 11.

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 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**. Subwindows, 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.

2.1 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 pressing the **Constrain 45** button in the **Attributes 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 45** button. 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 will cause it to 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). The **S button** to the left of the layer menu can be used to put *Xic* into layer-specific mode. In this mode, only objects on the current layer (the one highlighted in the layer table) will be selected. Otherwise, any object can be selected. All selected objects can be deselected with the **desel** button in the side menu.

Once selected, an object can be deleted, either by pressing the **Delete** key, or by pressing the **Delete** button in the **Edit 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 **Edit 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 **Edit 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. The layer-specific setting (from the **S button**) determines whether all objects are erased, or only those on the current layer. 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 in the **Edit 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** or from the list pop-ups in the **File Menu**. In addition, the **List** pop-ups contain a **Master** 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.

2.2 Electrical Schematic Editing

The eclectical 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 (11) 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 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.

2.2.1 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.

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 **Edit 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 **Edit 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** or from the List pop-ups in the **File Menu**. In addition, the **List** pop-ups contain a **Master** 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 **Current Transform** command in the **Edit menu**. Specifically, rotations are limited to multiples of 90 degrees, and there is no magnification capability.

2.2.2 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 vcc terminal device on the positive terminal of the voltage source.
4. Select the 'vcc' label of this terminal device.
5. Press the **label** button (side menu), and change the name from "vcc" 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 vcc 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.

2.2.3 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 precess, 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 **dots** command in the side 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 two special "devices" that can be used instead of wires to provide interconnections. These are the "gnd" and "vcc" devices. In the former 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 vcc terminal is more general purpose. As it is, this terminal will tie all locations attached to such terminals together. This, as the name suggests, is a convenient way of distributing a power net. If the name label of the vcc 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 "vcc" label of an existing vcc 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. Only wires on this layer are electrically active. Wires created on other layers are for decoration purposes only!

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 the active layer, and only this layer can be used for electrically significant wires and other objects. Additional 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 layer
NAME	device/subcircuit name property labels
MODL	device model property labels
VALU	device value property labels
PARM	device/subcircuit param property labels
NODE	vcc terminal label
ETC1	general purpose
ETC2	general purpose
TERM	general purpose

The **dots** button can be used to show dots at connection points. New users often appreciate the feedback provided by the **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 delete the vertex. Clicking on a vertex box with the **Shift** key held down will select that vertex, and allow the vertex to be dragged to a new location. 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.

2.2.4 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 (6.9) describing the **Properties** command in the **Edit Menu**. This section will describe some of the basic operations.

At this point there are three properties of interest: value, model, and param. Of these, 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 (such a specification would make no sense in in a SPICE context). The string for a device, which will be generated in SPICE output, has the generic form

device_name node_list model_or_value [*parameters*]

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 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.

2.2.5 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 the user will be prompted for a name for the terminal. The user can press **Enter** to accept a default name, or can enter a short descriptive name for the contact, such as "output".

Clicking on an existing terminal will delete the terminal. Clicking on an existing terminal with the **Shift** key held will "attach" the terminal to the mouse pointer, and clicking on a new location will move the terminal. Double clicking on a terminal with the **Shift** key held down will repeat prompting for the terminal's name.

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.

Once one or more terminals have been defined, the cell can "officially" be used as a subcell. Although cells without contact terminals can be placed, an error message will be generated, complaining that the subcell has no name property. It is possible, at this point, to use the **Push** command to push into the new cell, define the 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. The terminals will not disappear when clicked on. New terminals can only be added by returning to normal mode, and similarly the schematic can be edited only by returning to normal 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.

2.2.6 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 (4.14). Here, we will briefly outline its use. The node names can be edited only when the cell is *not* in symbolic mode, so that the **symbl** button must be disengaged if it is active. 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. It is very important that the assigned node name be unique in the circuit. This word will be used in SPICE output to designate the node, rather than the number.

The node mapping will be applied to the cell if the **nodmp** button was active when the cell was saved or last edited. Note that the **nodmp** button remains engaged after the node mapping editor is dismissed. This signals that the nodes will be mapped, i.e., the given names will be used. Press the **nodmp** button twice to pop up the node mapping editor again. If the **nodmp** button is disengaged, the default names will be used, and not the given names.

2.2.7 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 11.

2.3 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 **Edit** command or the **Place** command in the **Edit 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.

2.4 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*

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 .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 non-default 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
: The cell hierarchy will be written to a CGX file, whose name is the name of the cell suffixed with ".cgx". This file is produced if there are no errors. A conversion log file is also produced.

tocif
: The cell hierarchy will be written to a CIF file, whose name is the name of the cell suffixed with ".cif". This file is produced if there are no errors. A conversion log file is also produced.

togds
: The cell hierarchy will be written to a GDS file, whose name is the name of the cell suffixed with ".gds". This file is produced if there are no errors. A conversion log file is also produced.

tooas
: The cell hierarchy will be written to an OASIS file, whose name is the name of the cell suffixed with ".oas". This file is produced if there are no errors. A conversion log file is also produced.

toxic
: The hierarchy will be written out as native symbol files. This makes sense if a GDSII or CIF file is given on the command line. A conversion log file is also produced.

drc
: Design rule checking is performed over the entire cell, and results are written to a log file.

In batch mode, the log files for reading and writing of files are written to the current directory.

2.5 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           3002/tcp   #Whiteley Research Inc.
```

where the port number 3002 is replaced by the desired port number. If there is no port assigned for "`xic/tcp`", port 3002 is used.

If the environment variable XTNETDEBUG is defined, *Xic* will not go into the background, but will service requests while printing status messages to the standard output. This may be useful for debugging.

The user's application should open a socket to this port for communications. Only one process can access the port at a given time. For testing and experimental purposes, the telnet program can be used to connect to the *Xic* daemon. 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.

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 default data mode for communications is `ascii` mode. In this mode, all data are transferred as ASCII text. Also available is `binary` mode, where data returned by the server is in a binary format, which can be more efficient for large data transfers.

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. The modes implied by these commands apply to the current connection, and never carry over to subsequent connections. These commands consist of a single word from the table below:

`close`
: This will close the connection to the daemon, and is the normal way to end a session. 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 begin 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.

`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.

`ascii`
: When given, subsequent transmissions will use the ascii data format. This is the default format.

`binary`
When given, subsequent transmissions will use the binary data format.

`longform`
When given, subsequent transmissions will use the long form for returned data, meaning that the actual data are returned.

`shortform`
When given, subsequent transmissions 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 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.

`dieonerror`
Ordinarily, if the client crashes or there is a connection failure, the server will simply clear itself and wait for new connections. If `dieonerror` was given, the server will instead exit.

`nodieonerror`
This will undo the effect of `dieonerror`, if `dieonerror` was given, and has no effect otherwise.

`keepall`
Ordinarily, when the server receives a `close` command, 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.

`nokeepall`
This will undo the effect of `keepall`, if `keepall` was given, and has no effect otherwise.

The normal way to terminate a session with the server is to issue the `close` command, which will reset and clear the server and close the communication channel, and the server will begin listening for new connections. The "reset and clear" is equivalent to giving the `reset` and `clear` commands. 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 is always reset and cleared, or will exit if `dieonerror` was in effect.

When a new connection is established, all of the modes associated with the commands assume the default values. These states do not carry over if `keepall` was given, so that the appropriate commands must be repeated to establish the same state.

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.

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.

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.

If the XTNETDEBUG environment variable is set when *Xic* is started in server mode, a debugging mode is active, and *Xic* remains in the foreground. 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*.

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.

2.5.1 The Ascii Format

This is the default format. If a result is obtained from the line processed, that result is sent as an ASCII string. The possible return message strings are given below. In the table, the '\r' represents the ASCII line return character, and '\n' represents the ASCII line feed character. Other characters are verbatim, unless italicized, in which case they represent data. All returns end with the characters "\r\n", with no trailing null byte.

"ok> \r\n"
: The line was processed, and the next line can be supplied.

"-?> \r\n"
: The line was processed and a block is open. This is given when entering internal lines in a `while` block, for example, before the `end` token. The block is not executed until the `end` is received. The same applies between `function` and `endfunc`.

"error> \r\n"
: The input just given produced an error.

"*datatype data*\r\n"
: The command produced data, which is (optionally) returned (see below).

If there are data returned, the first returned token is a keyword giving the type of data. In the default "shortform" mode, this is immediately followed by "\r\n" and there is no further transmission. In "longform" mode, the type name is followed by a single space character, then the printed representation of the data, terminated with "\r\n". The following table gives the data type keywords and the "longform" format. Printed tokens are separated by a single space character. When receiving data, if the first character following the data type keyword is not a space, the "shortform" mode is in effect.

Data Type	Description	Long Form
`scalar`	Single floating-point value	"`scalar` *value*\r\n"
`string`	Text string	"`string` *string*\r\n"
`array`	Multiple floating-point values	"`array size value` ...\r\n"
`zlist`	List of trapezoids	"`zlist` *num xll xlr yl xul xur yu* ...\r\n"
`lexpr`	Layer expression (string)	"`lexpr` *string*\r\n"
`handle`	Handle to an object	"`handle` *id*\r\n"

In "longform" mode, following the keyword and a single space are the data, in printable ASCII format: a numeric value for `scalar`, an unquoted string for `string`, or the size of an array followed by the printed array following `array`. Note that the array returns the allocated size, which is not necessarily the size in use. For a trapezoid list, the number of trapezoids is printed, followed by six values for each trapezoid (lower-left x, lower-right x, lower y, upper-left x, upper-right x, upper y). Layer expressions are returned as a string. Handles are returned as the internal integer index, which is given as a string. The return data are followed by carriage return and line feed characters. For string data, these must be stripped from the string data returned to retrieve the actual string sent.

2.5.2 The Binary Format

This format is used if the `binary` command is in effect. It is somewhat more efficient and easier to parse. 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 possible return values are listed in the table below. These correspond directly to the ASCII returns described in the previous section. The first 4-byte field indicates the type of return. If the `longform` mode is in effect, the data are returned in the format shown below, otherwise there is no further content.

ok	(integer 0)	
in block	(integer 1)	
error	(integer 2)	
scalar	(integer 3)	(double *value*)
string	(integer 4)	(integer *length*) (*length*-byte string)
array	(integer 5)	(integer *size*) (*size* 8-byte values)
zlist	(integer 6)	(integer *num*) (*num**6 4-byte integers)
lexpr	(integer 7)	(integer *length*) (*length*-byte string)
handle	(integer 8)	(integer *id*)

The first three responses correspond to the "ok", "in block" and "error" return prompts, respectively. The last six responses correspond to data types. For scalar data, a double precision (8-byte) data item follows. For strings, arrays, zlists and lexprs, a size integer follows, followed by the data. For strings, the size includes the null-byte string termination. For arrays, the size is the number of double precision values (8 bytes each) that follow. For trapezoid lists, te size is the number of six-integer trapezoid fields that follow. The handle return is a single integer giving the internal id.

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 routine 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.

2.5.3 Operation

Internal script variables are defined and set in accord with instructions received. The variables and other context are cleared when a connection to the server is made or broken, 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.

2.6 Template (Parameterized) Cells

Template cells, or parameterized cells, are cells which in addition to possibly containing fixed geometry, contain a script which creates geometry. When a template cell is instantiated, parameters are supplied to the script, so that the geometry can vary between instantiations. A single template cell can replace numerous small cells representing contacts, vias, and devices. Use of template cells can streamline the design process and reduce errors.

Template cells differ from ordinary cells in the following ways:

- The name of a template cell must be in the form

  cellname`XXX`

  where the *cellname* is arbitrary, but must be followed by the characters "`XXX`" (three upper-case 'X' characters). The "`XXX`" is replaced internally by an identifying code which is used to distinguish between instances with different parameter sets.

- Template cells have a special property, the Template Script property, which is assigned property number 7199. The string of this property is a script that will be executed when the cell is instantiated. The format of the script is conventional, and any of the geometry creation and related functions can be used.

- Template cells may have an optional Template Params property which is assigned number 7198. The string of this property is a comma separated list of assignments, e.g.,

  ```
  length=100,width=20,bottom=2,layer=R1
  ```

Each of the `length`, `width`, etc. are arbitrary parameter names, that is, names of variables used in the script, but not generally defined in the script. The values are taken as default values for the parameters, and can be numeric values or strings. If a value string contains white space, it must be double quoted (`"like this"`). If the value token is a string constant, i.e., the double quotes should be preserved, then the value should be both single and double quoted (`'"like this"'`). 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"
```

Either the physical or electrical part of a symbol can be a template cell, or both. A template cell can contain geometry, subcells, and properties just like any other cell.

2.6.1 How Template Cells Work

A template cell is never itself instantiated. When one places an instance of a template cell, the following steps occur:

1. The template cell is read into memory if it is not already there.
2. The user is prompted for the parameter values, if any.
3. The database is searched for another cell derived from the same template with the same parameter values. If one is found, it is instantiated, and we're done.
4. Otherwise, the template cell is duplicated, and the "`XXX`" is replaced with a unique identifier in the new cell. The script is executed, and the Template Script property is removed from the new cell. A property is added to the new cell identifying the template (the Template Name property, value 7197). The new cell is instantiated, and we're done.

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
```

Note that the double quotes needed to hide white space and commas are stripped. To preserve double quotes, as would be necessary for a string constant, the double quoted entry should additionally be single quoted, as for the `name` in the example.

Once the instance is placed, it behaves in all respects as a normal cell. It has a "master" derived from the template, and a unique master exists for each unique parameter set. Writing the hierarchy to GDSII of CIF produces a perfectly normal file. The template cells are never included in the output file, since they are not directly instantiated in the hierarchy.

2.6.2 Creation of a Template Cell

To create a template cell, one can follow this procedure:

1. Write a script that creates the geometry desired in the template. The script can be authored as any other script. It should be thoroughly debugged before committing it to a template cell.
2. Use the **Edit** command to edit a new cell which will become the template cell. Remember that the name of a template cell must be suffixed with "`XXX`".
3. Add any geometry to the cell that is necessary. This can be done at any time.
4. Bring up the **Cell Properties** editor.
5. Press **Add**, which brings up a pop-up menu, and select **Tmpl Script** in the pop-up menu.
6. Press the "**L**" button to the left of the prompt line. This brings up the **Text Editor** pop-up.
7. Use the **Read** button of the text editor (in the **File** menu) to read in the script file. Perform any last minute editing.
8. Press the **Save** button in the text editor. The text editor will go away, and the script will have been saved in the current cell.
9. In the **Cell Properties** editor, press **Add**, the select **Tmpl Params** in the pop-up menu.
10. Enter a string on the prompt line consisting of each parameter name followed by '=' and its value. The *name* = *value* pairs can be separated by commas or white space. If a *value* contains white space, commas, or an equal sign, it should be quoted by double quotes ('`"`'). If the value is a string constant, it should be quoted with both single and double quotes, as in `Param1 = '"this is a string"'`. Press **Enter** when done.
11. This completes the operation. Save the cell to disk.

2.6.3 Adding an Instance of a Template Cell

1. From the **Cell Placement Control** panel, set the current master to the name of a template cell, which should exist in the search path.
2. Press the **Place** button. A special indicating icon will be "attached" to the mouse pointer.
3. Click with button 1 in a drawing window. The instance will be placed at that location. If the template has parameters, the parameter string will be shown on the prompt line. The values are the default values for the template, and the user is asked to edit the line to provide the desired parameters.
4. Edit the parameter values, and press **Enter** when done, or **Esc** to abort the placement.

   The parameter string is shown in the form as saved in the cell, i.e., with all delimiting white space removed and commas separating the *name* = *value* pairs. The string entered can be more loosely formatted, with white space added for readability and the separating commas optional. However, giving an unrecognized parameter name or syntax error will abort the operation.
5. Press **Enter** to supply the parameters. The instance using these parameters will be created and placed (the current transform is used in the placement, as for normal cells).

6. The process can be repeated to place additional instances, where the user clicks with button 1. In this case, the parameter string presented for modification is the last string entered, and not the default string from the template.

Template cells can also be added from within a script, which may be running in batch mode where user prompting is not available. The `PlaceTemplateArgs` script function is used to set the parameter string used for instantiation. This overrides the default parameters set in the template cell itself, and the saved parameter set will be used for all instantiations of the template, until changed with another call to `PlaceTemplateArgs`. The actual placement is done with the `Place` script function, as for normal cells, however the cell name is the name of the template cell.

For example:

```
PlaceTemplateArgs("boxXXX", "width=2,height=5")
Place("boxXXX", 10, 25)
PlaceTemplateArgs("boxXXX", "width=4,height=5")
Place("boxXXX", 10, 50)
Place("boxXXX", 10, 75)
```

This will place three instances of the template cell "`boxXXX`", the first with parameter set "`width=2,height=5`" and the others with parameter set "`width=4,height=5`". The `PlaceTemplateArgs` function sets the default parameter string shown when prompting for parameters in graphical mode, which is automatically applied when there is no prompting.

After creating new template instances, one or more new template-derived master cells will exist in memory. These cells must be written to disk to enable subsequent use of the cell containing the new instances. This is automatic if the current cell is saved in an archive format, as the entire cell hierarchy is written to the file. If saving as native cell files, the `UpdateNative` script function is a convenient way to save the current cell and any new or modified subcells to cell files on disk.

2.6.4 Changing the Parameters of an Instance

Once a template cell has been instantiated, the instance can be changed to represent a new set of parameter values **if** the template cell is available. Thus, when a design is exported to another site that may wish to modify the cell parameters, the template cells must be exported as well. The template cells are **not** automatically added to GDSII files or the other file formats. They should be supplied as *Xic* cells, in addition to the GDSII or other output.

One convenient way to maintain template cells is to place them in a library.

Assuming that the template cell is available, the following procedure can be used to alter an instance to a new parameter set.

1. Select the instance to modify.
2. Bring up the **Property Editor** from the **Edit Menu**. The editor will show a list of the properties and pseudo-properties of the instance.
3. In the listing, look for a property number 7198 named "`Tmpl Params`". If this does not exist, the cell is not an instance of a template, or the template has no parameters.

4. Click on the `Tmpl Params` property in the list, and press the **Edit** button in the **Property Editor**.
5. The current parameter assignments will be listed on the prompt line, and the user can perform the modifications.
6. Press **Enter** after the new parameters have been given. The instance will be rebuilt with the new parameter values.

2.6.5 Changing the Parameters of a Template-Derived Master

One can change all of the instances that use a particular parameter set to a new parameter set by changing the master cell of the instances. The original template cell must be accessible, as for changing individual instances. The procedure is as follows.

1. Make the master cell the current cell. The easiest way to do this is to select an instance of the master cell (one of the instances to be changed) in a drawing window, and use the **Push** command in the **Edit Menu**. The master can also be selected for editing from the **Cells Listing** pop-up, or by giving its name in the **Edit** command.
2. Press the **Cell Properties** button in the **Edit** menu. This brings up the **Cell Property Editor**, which will list the properties of the current cell (the master derived from a template).
3. In the listing, there should be an entry for property number 7198 named "`Tmpl Params`". If not, the current cell is not a master derived from a template, or the template has no parameters.
4. Select the `Tmpl Params` property, and press the **Edit** button in the **Cell Property Editor**.
5. The current parameter string will appear on the prompt line, where it can be edited by the user. Edit the string to supply the new parameters.
6. Press **Enter** when the modifications are complete. This will rebuild the master from the new parameter set. All instances of this master will contain the new master's geometry.

2.7 Cadence^{TM} Compatibility

PRELIMINARY

The first shot at a package for limited compatibility with products from Cadence Design Systems Inc., is now available, through the technology file keyword:

ReadCds *filename*

This can appear anywhere in the technology file, and will cause *Xic* to read information from the Cadence startup file given in *filename*. This should be a full path to the file, unless the file is in the library search path.

Presently, there are two types of Cadence startup files that are understood: the **Design Resource File**, and the ASCII **Technology File**. Although the names may vary, the design resource file in one installation is named "`display.drf`" and the technology file is named "`techfile.txt`". It is also possible that the data are combined into a single file. In this case, the design resource information should come first.

The design resource file must be read first, i.e., there should be two calls to ReadCds, first with the design resource file, then for the technology file. A minimalist *Xic* technology file can consist of these two statements only, or just one statement if the files are combined. This will set the layers and their colors, fill patterns, and some of the electrical information.

When a technology file is written with the **Save Tech** command, it will have the usual format and the ReadCds lines are *not* included in the new file.

The files are collections of "nodes", as understood by the Lisp parser (see 14.16.3). A named node has the form

name(*data* ...)

The *data* are other Lisp nodes, strings, or numerical data or expressions. This can occupy arbitrarily many lines in the file. There can be no space between the node name and the opening parenthesis.

The files consist of one or more successive Lisp nodes, with names that are defined by Cadence. A node which consists of a list of nodes is termed a "class". Most of the top-level nodes which appear in the ASCII technology file are actually classes. The nodes that are defined by Cadence for these two files, and are understood by *Xic* are described in the sections below.

2.7.1 Design Resource File

The following top-level design 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 design resource nodes has no effect on *Xic* operation.

`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. Patterns with 32X32 map size are ignored.

`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.

2.7.2 Technology File

The following nodes and classes are read from the Cadence ASCII technology file.

`include`
This node contains a string, which is a path to another Lisp file. That file will be opened and read.

`comment`
This node is ignored.

`controls`
This class contains the following nodes.

`techParams`
This node is ignored.

`techPermissions`
This node is ignored.

`layerDefinitions`
This class contains the following nodes.

`techLayers`
A new layer is created in the *Xic* layer table for each entry with a layer number in the range 0–127. The abbreviation field is ignored. The layer "long name" will be assigned the layer name, if it is more that four characters long. New layers are created in the order of appearance.

`techPurposes`
This node is ignored.

`techLayerPurposePriorities`
This will reorder the layers already in the *Xic* layer table in the order given. Only layers with the purpose "`drawing`" are considered. This should appear after `techLayers`.

`techDisplays`
This will assign the colors and fill patterns to layers that exist in the *Xic* layer table. Only entries with purpose "`drawing`" are considered. This references the internal packet, color, and stipple lists created from the design resource nodes.

`techLayerproperties`
This node is ignored.

`layerRules`
This class contains the following nodes.

`viaLayers`
The conducting layers are assigned the Conductor attribute. The via layer is assigned the Via and DarkField attributes.

`equivalentLayers`
This node is ignored.

`streamLayers`
A GDSII import/export mapping is applied for each layer given.

`devices`
This class contains the following nodes.

`tcCreateCDSDeviceClass`
This node is ignored.

`symContactDevice`
This node is ignored.

`ruleContactDevice`
This node is ignored.

`symEnhancementDevice`
This node is ignored.

`symDepletionDevice`
This node is ignored.

`symPinDevice`
This node is ignored.

`symRectPinDevice`
This node is ignored.

`tcCreateDeviceClass`
This node is ignored.

`tcDeclareDevice`
This node is ignored.

`physicalRules`
This class contains the following nodes.

`orderedSpacingRules`
This node is ignored.

`spacingRules`
This node is ignored.

`mfgGridResolution`
This node is ignored.

`electricalRules`
This class contains the following nodes.

`characterizationRules`
This node is ignored.

`orderedCharacterizationRules`
This node is ignored.

`leRules`
This class contains the following nodes.

`leLswLayers`
This node is ignored.

`lxRules`
This class contains the following nodes.

`lxExtractLayers`
This node is ignored.

`lxNoOverlapLayers`
This node is ignored.

`lxMPPTemplates`
This node is ignored.

`compactorRules`
This class contains the following nodes.

`compactorLayers`
This node is ignored.

`symWires`
This node is ignored.

`symRules`
This node is ignored.

`lasRules`
This class contains the following nodes.

`lasLayers`
This node is ignored.

`lasDevices`
This node is ignored.

`lasWires`
This node is ignored.

`lasProperties`
This node is ignored.

`prRules`
This class contains the following nodes.

`prRoutingLayers`
Each layer given has the Conductor and Routing attributes applied.

`prViaTypes`
This node is ignored.

`prStackVias`
This node is ignored.

`prMastersliceLayers`
This node is ignored.

`prViaRules`
This node is ignored.

`prGenViaRules`
This node is ignored.

`prTurnViaRules`
This node is ignored.

`prNonDefaultRules`
This node is ignored.

`prRoutingPitch`
This node is ignored.

`prRoutingOffset`
This node is ignored.

`prOverlapLayer`
This node is ignored.

This page intentionally left blank.

Chapter 3

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
About	`about`	**About Panel**	Show version info
Release Notes	`notes`	**Text Editor**	Show release notes

3.1 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.

Under Windows, the help viewer and HTML-based info windows use a Microsoft technology called OLE (Object Linking and Embedding) and the Internet Explorer HTML viewer. It actually uses the internals of Internet Explorer to do the HTML rendering. This requires that Internet Explorer 4.0 or newer be installed. Although this technology is admittedly rather nifty, it has rather amazing internal complexity, and when all is said and done, sad and severe limitations. Although the basic functionality is provided, many of the more advanced/obscure features found in the Unix/Linux help system are missing.

If the boolean variable NoHelpDefault is set (with the **!set** command), 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.

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 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. The name given to the **!help** command, or to 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.
- An arbitrary URL accessible through the Internet.

If the given name can be resolved, the resulting text 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*. The prefix "`http://`" *must* be provided with the URL. Thus, for example,

```
!help http://www.wrcad.com
```

will bring up the Whiteley Research web page. 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.

The URL given in the **Open** command or through the **!help** command 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`", "`.doc`", "`.log`", "`.scr`", "`.sh`", "`.c`", "`.cc`", and "`.h`".

Holding **Shift** while clicking on an anchor that points to a URL which specifies a file on a remote system will download the file, as in Netscape. Downloading makes use of the `httpget` utility program available in the Accessories distribution. Installation of the accessories is required for downloading to be available under Unix/Linux. 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 pop-up 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.

3.1.1 The HTML Viewer

When a new help window is about to appear, initialization is provided by a file named "`.mozyrc`" in the user's home directory, if it exists. This file sets a number of defaults used in the viewer, and can be customized by the user. If the file is not found, internal defaults are used. A sample initialization file is provided with the distribution, and can be installed by the user. This is described in more detail in 3.1.4.

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 **Open** button in the **File** menu pops up a dialog into which a new keyword, URL, or file name can be entered. 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.

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.

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.

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.

The **Quit** button in the **File** menu removes the help window, and will exit help mode if there are no other help windows visible. In *Xic*, 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 **Search** 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".

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 the **Find Text** button or the **Enter** key. The text is searched from the top line in view or the line following an existing selection, toward the end of text. Additional matches can be found by pressing the **Find Text** button repeatedly. The search is case-insensitive.

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. The latter does not actually have to be fixed-pitch, however preformatted columns will not line up unless this is so. Pressing **Apply Font** will set the currently selected font as the viewer's proportional font, and **Apply Fixed Font** will set the currently selected font as the viewers "fixed" font. In either case, the display will be redrawn using the new font. The new font selection will be updated in the `.mozyrc` initialization file if it exists, so that subsequent windows will use the new font.

The **Filter** page in the pop-up can be used to limit the choices. In particular, if you want a large font, selecting only **Scalable** in **Font Types** eliminates the ragged-looking bitmap fonts from the selection. One can also limit the choices to monospaced fonts when selecting the fixed-pitch font.

In *Xic*, there are commands to set the font families:

```
!helpfixed [family-size]
!helpfont [family-size]
```

The *family-name* is the X Font Family Description with "-*size*" appended. The defaults are:

fixed	`fixed-misc-normal-*-14`
proportional	`adobe-times-normal-p-14`

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.

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. There is a **No Cookies** button in the help window **Options** menu to disable sending and receiving cookies. The NoCookies keyword in the `.mozyrc` file can be set to 1 to disable cookies by default, or 0 for the default (enabled) behavior. With cookies, it is possible to view the New York Times web site, for example, which requires registration. 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 info like credit card numbers, though.

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 a 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.

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.

If the **Bad HTML Warnings** button in the **Options** menu is active, messages about incorrect HTML format are emitted to standard output.

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.

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.

3.1.2 The Help Database

The 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/lib/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 A.7.

3.1.3 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 13.11.

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.

3.1.4 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 never created, but if it exists it may be updated by certain operations, such as when fonts are saved. This makes the change persistent. A sample `.mozyrc` file is provided in the startup directory of the *Xic* and *WRspice* distributions. See the comments in the file for the defaults that can be changed. One should place this file as "`.mozyrc`" in the home directory to change the defaults.

Incidently "mozy" is the name of the stand-alone version of the HTML viewer/web browser available on the Whiteley Research web site.

3.2 The About Button: Program and Legal Info

The **About** command 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.

3.3 The Release Notes Button: View Release Notes

This command 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/lib/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.

This page intentionally left blank.

Chapter 4

The Side Menu: Geometry Creation

Xic has a "side" menu of buttons, typically displayed to the left of the main window, though if the environment variable XIC_MENU_RIGHT is set when *Xic* starts, the menu will be placed to the right of the main drawing window. This section describes in detail the commands available in the side menu in physical and electrical modes. These include all of the commands for geometry creation, and other frequently used commands.

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 "keys" box just below the side menu buttons in the main window, or in the upper-right corner of subwindow 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 **Current Transform** button in the **Edit menu**.

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 side 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 14.

The table below summarizes 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.

Physical Side Menu		
Icon	Name	Function
	`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
	`desel`	Deselect objects
	`layer`	Set selection mode

Electrical Side Menu		
Icon	Name	Function
	`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`	Set node name mapping
	`subct`	Set subcircuit contacts
	`terms`	Show terminals
	`dots`	Show contact dots
	`run`	Run *WRspice*
	`deck`	Save SPICE file
	`plot`	Plot SPICE results
	`iplot`	Set dynamic plotting
	`desel`	Deselect objects
	`layer`	Set selection mode

Table 4.1: Commands found in the side menu in physical and electrical modes.

In addition, the following buttons are associated with the layer menu.

Name	Function
`lspec`	**S button**, set layer-specific mode
`ltup`	Up-arrow, forward scroll layer table
`ltdn`	Down-arrow, backward scroll layer table

4.1 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 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 disabled by setting the NoSnapToEdges variable.

In electrical mode, an arc is actually a wire, and as such should not be used on the SCED layer. Although there is no error, the arc vertices are considered in the connectivity establishment, leading to inefficiency. An arc created with "SCED" as the current layer will actually be assigned to the next layer. If the user insists on the arc being on the SCED layer, the **Chg Layer** command in the **Edit 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.

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.

4.2 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.

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 disabled by setting the NoSnapToEdges variable.

In physical mode, if an electrical specification exists for the current layer, as set in the technology file (see 11.1.1) the electrical parameters for the box being created can be printed by holding the **Shift** key while releasing or clicking mouse button 1, whichever would complete the box. In this case, the box is not created, but the electrical parameters for the box are displayed. This allows the dimensions of the box to be fine-tuned before actual creation. An electrical specification for the layer provides computation of resistance, capacitance, or transmission line parameters for boxes on the layer. See 11.1.1 for more information.

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. On layers other than the SCED layer, the box is rendered with a dotted border and no fill. On the SCED layer, the fill attribute associated with the layer is used. If a fill or solid border is desired on an upper layer, a polygon should be used rather than a box. Boxes have no electrical significance, but can be used for illustrative purposes.

4.3 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 disabled by setting the NoSnapToEdges variable.

4.4 The deck Button: Dump 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:

- 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.

Node numbers will be assigned internally, unless the **nodmp** button is active. In this case, user-specified names and the labels of "vcc" devices will be used as node tokens rather than the assigned integers. See 4.14.1 for information on node mapping.

If the variable CheckSolitary is set (with the **!set** command) then a warning will be issued if nodes are found with only one connection.

4.5 The desel Button: Deselect Objects

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. See 1.3.10 for more information on how selection is performed.

4.6 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 two types of menu available, the default being a character-keyed pulldown menu, and the alternative being a pictorial menu showing the device symbols. The pulldown menu consumes less screen space.

One can specify default use of the pictorial menu by setting the variable PictorialDevs with the **!set** command (enter"`!set PictorialDevs`"). The **!unset** command can be used to revert to the default

menu. The new menu appears the next time the menu is popped up, after the variable is changed. In either case, buttons on the device menus allow switching between the two styles.

The default menu consists of a sequence of buttons containing the first letter of the device name (not the SPICE key). Pressing and holding button 1 while the pointer is over one of these buttons will bring down a menu of the device names that start with that letter. Moving the pointer down the menu will highlight the entry under the pointer. A selection is made by releasing the button. In the pictorial menu, the devices are shown in alphabetical order. Clicking on one of the devices will establish the selection.

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 **Current Transform** button in the **Edit Menu**.

The devices available and other details depend upon the definitions in the device library file (see A.5). 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. Information below pertains to the default device library file.

The following table lists the devices found in the default device library file.

Name	Description
Contact Devices	
gnd	Ground Contact
vcc	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
opamp	Example Macro

The colors used in the pictorial device menu can be changed by setting variables. These are

variable	purpose	default
pop_color_dvbg	background	`gray90`
pop_color_dvfg	foreground	`black`
pop_color_dvhl	highlight	`blue`
pop_color_dvsl	selection	`gray80`

4.6.1 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.

Vcc Device

The vcc device has an associated label (defaulting to "vcc") which can be changed by the user by selecting the label and pressing the **label** button in the side menu. All nodes connected to a vcc device with the same label are taken as being connected together. The label of the vcc device is also used to name the net when node mapping is active, as set with the **nodmp** button in the side menu.

This will not tie nodes between the main circuit and subcircuits, or between subcircuits, unless the terminal name is also defined in a `.global` line for SPICE. If not global, then it will only tie together the nodes of each subcircuit, and the main circuit, due to the modification of node names that occurs during subcircuit expansion. The `.global` line prevents this modification, and can be added as a `spicetext` label.

4.6.2 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.

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 vcc terminal 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 vcc terminal 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 value and/or param property should be added to specify the controlling voltage source or inductor, 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 value and/or param property should be added to specify the controlling voltage source or inductor, 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 model property should be added to specify the controlling voltage source or inductor, and a 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.

4.7 The donut Button: Create Donut Object

The **donut** button appears only in physical mode. It is used to create a 360 degree arc with a hole. 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.

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.

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 disabled by setting the NoSnapToEdges variable.

If the SpotSize variable is set to a value (with the **!set** command), the figure is constructed somewhat differently. With the SpotSize variable set to 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, 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. This conditioning, with SpotSize set nonzero, applies only to objects created with the **round** and **donut** commands, and not the **arc** command or general polygons.

4.8 The dots Button: Show Connections

The **dots** button is available in electrical mode. While active, visually ambiguous connection points are marked with dot. A yes/no box appears asking whether to place dots at all connections or not. If “yes” is selected, all connections will be shown with a dot. If “no” is given, dots are placed at 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.

The ShowDots variable also controls the display of dots. This variable is internally set by the **dots** button, and setting/resetting the variable will change the status of the menu button accordingly. If set to a word that begins with ”a”, all connection points are marked with a dot. If set to anything else, only the potentially ambiguous points are marked, and if unset, no dots are shown.

4.9 The erase Button: Erase Geometry

Rectangular regions of polygons, boxes, and wires can be erased with the **erase** button. The user clicks twice or presses and drags to define the diagonal of the region to be erased. If in layer-specific mode (the **S button** is active), only the current layer is erased, otherwise all layers are 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.

If the **Ctrl** key is held while the box is completed, the objects under the box will be clipped to the box, rather than being clipped around the box.

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 disabled by setting the NoSnapToEdges variable.

When objects are erased in physical mode, they are added to the yank buffer chain. The yank buffer chain has a depth of five, meaning that the contents of the last five erasures are available for placement with the **put** command.

If the **Shift** key is held during the erase operation, there is no erasure, however the pieces that would have been erased are added to the yank buffer. If the **Ctrl** key is also held, the effect is the same, except that the saved pieces are from outside of the box.

When the **Ctrl** key is held before the box is defined, clicking on a subcell will erase objects within the bounding box of the subcell. This is the inverse of the subcell paint operation in the **box** command.

4.10 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.

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**. The trace points are saved along with the file, as with the **plot** command, so the plotting commands will be persistent.

4.11 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. In this mode, the command exits after the new text is entered on the prompt line. The **Enter** key terminates the label text.

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 **Current Transform** button in the **Edit Menu**. Instances of the label are placed where the user clicks in a drawing window.

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%

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 **Attributes Menu** is set, 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.

4.11.1 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.

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**.

4.11.2 Spicetext Labels

In electrical mode, labels can be used to enter arbitrary text into the SPICE output. If the first word of the label is of the form

`spicetext`*N*

the text which follows will be entered verbatim as a separate line in the SPICE output. 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 in this manner, but erroneous syntax will of course cause errors as the SPICE text is sourced.

4.11.3 Hypertext

Xic uses a "hypertext" mechanism in the representation of property strings and label text in electrical mode. 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.

This same capability applies when adding or editing properties from the **Property Editor** provided by the **Properties** button in the **Edit Menu**.

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 is not 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.

The hypertext capability is active when editing a string for a property or label in electrical mode. One inserts a hypertext reference into the text shown on the prompt line by clicking with button 1 in a drawing window, over the object to be referenced. There are three types of reference:

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.

This feature can be used to set up specialized SPICE output. Suppose one wishes to use the **save** command 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** command 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.

4.11.4 "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 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.

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.

4.11.5 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.

4.11.6 Label Size Issues

In electrical mode, property text labels with text size longer than a certain length are not displayed; a small box at the text origin is displayed instead. The label text can be shrunk (with the **Stretch** command in the **Edit Menu** or with button 1 operations) to make it visible. The **Info** command in the **View Menu** can be used to see the text of a non-visible label.

Non-visible 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 variable MaxPrpLabelLen. This can be set to an integer greater than 6 with the **!set** command, and 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.

A better 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.

4.11.7 Label Font File

It is possible to change the font used for labels and the **logo** command. The default font is set internally by *Xic*, however individual characters or the whole font used for labels will be updated upon startup if a file named "`xic_font`" is found along the library search path, which contains alternative character specifications.

The font file consists of vector specifications for the characters '`!`' through '`~`' in the ASCII chart. These are stored in a compact scalable form. The `xic_font` file consists of character specifications of the form described below. 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.

The existing font can be dumped to a file with the command

`!dumpfont [`*filename*`]`

If *filename* is absent, the font will be dumped to "`xic_font`" in the current directory. Any existing file of the same name is saved with a "`.bak`" extension.

Similarly, the font used with the **logo** command can be altered with a "`xic_logofont`" file, which can be created with the **!dumplogofont** command.

4.12 The layer Button: Set Selection Mode

The **layer** in the side menu produces the **Selection Control Panel** which provides a number of mode switches which control object selection.

The **Layer Specific** button sets whether *Xic* is in layer-specific mode. Layer-specific mode can also be set with the **S button** to the left of the layer menu.

In layer-specific mode, only objects on the current layer, or subcells containing objects on the current layer, can be selected and deselected with mouse operations. Otherwise, any visible object can be be

selected or deselected. Instances are selected during a layer specific point select if there is no other qualifying geometry where the user clicked, and only instances that contain objects on the current layer can be selected in layer-specific mode. The layer-specific mode affects many of the commands and operations in *Xic*, typically limiting the operation to objects on the current layer when active.

Layer-specific mode is indicated by a small box icon in the far right side of the layer table. Clicking on this icon with button 1 will also toggle layer-specific mode on and off.

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.

4.13 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. The minimum width of the wire is the MinWidth design rule for the layer.

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 which constructs the characters using Manhattan polygons. The "pretty" font is one of the system fonts, which similarly creates characters constructed as Manhattan polygons.

The font used can be specified by setting variables, or more conveniently through use of the **!logofont** command. The **!logofont** command brings up the **Font Selector** pop-up, allowing the user to select a "pretty" font for use with the **logo** command. It also allows the user to select use of the internal (vector) font, or the Manhattan font. The buttons along the bottom of the pop-up accomplish this, and the label at the bottom indicates the current selection.

The Manhattan font can also be enabled for the creation of new physical labels by setting the variable LogoManhattan with the **!set** command, i.e.,

```
!set LogoManhattan
```

The minimum size for a "pixel" in this and the "pretty" font is the MinWidth design rule parameter. The Manhattan font is fixed-pitch with an 8X16 map.

Use of the "pretty" font is enabled by setting the boolean variable LogoPretty. This variable, and the LogoManhattan variable, can be set through the **!logofont** command. When LogoPretty is set, the **logo** command will use a font derived from the screen fonts supplied by the system. The LogoPretty variable overrides the LogoManhattan variable, if both are set. When LogoPretty is set, the **logo** command will use the font named in the LogoPrettyFont variable, or a default font if this variable is not set. This variable can be set by hand, or can be set from the **!logofont** 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.

When using the default vector font, there is a path width parameter which takes integer values from 1 through 5. A larger parameter produces a thicker path, making the characters appear increasingly bold. The value of this parameter can be set with the LogoPathWidth variable, e.g.,

```
!set LogoPathWidth 3
```

The default value is 1. This parameter applies only to the vector font, and has no effect when rendering the Manhattan or pretty fonts. 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.

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/lib/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.

The minimum size of a character cell is about .8 X 1.4 microns (design rules permitting) for the vector font, 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.

The default operation is to apply the text or image feature directly in the current cell, where the user clicks. Previous versions of this command instead created a subcell containing the text, which was instantiated at the clicked-on locations. This may be more efficient if there are many copies of the same label. This operating mode can be enabled by setting the variable LogoToFile with the **!set** command.

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.

Note that although the character path width is MinWidth or larger, some characters may produce design rule violations due to acute angles. 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.

The vector font used to render the characters can be modified, through use of a font file named "`xic_logofont`". This has syntax identical to the "`xic_font`" file described with the **label** command. The existing logo font can be dumped to a file with the **!dumplogofont** command. Also, the variable LogoEndStyle can be set to change the default end style used for the paths. The variable should be set to an integer 0–2, where 0 is for flush ends, 1 for rounded ends, and 2 for extended ends. This variable is ignored if the vector font is not in use.

4.14 The nodmp Button: Enable SPICE Node Mapping

When this button, which is available only in electrical mode, is active, SPICE output will contain text tokens instead of the internally generated numerical node names. This provides predictability of the node names, as the internally generated node names can change if the circuit is modified. The state of this button is saved with the cell, and is set when a new cell is opened for editing.

When this button is activated via a button press, the **Node Mapping** panel appears. Since this button is set from the saved state in the current cell, one may have to turn the button off, then on again to bring up the panel. This panel provides control of the mapping.

The panel allows arbitrary names to be applied to the nodes. When mapping is active, the default names for the nodes that correspond to the contact terminals of the cell are the names of the terminals. The terminal names are set when the terminals are defined with the **subct** command, and it is also possible to name terminals from the **Edit Terminals** command in the **Extract Menu**.

Though the mapping of the current cell can be edited whenever the panel is visible, the mapped names are used in SPICE output only when the side menu button is active, or in subcells where the button was active when that cell was last edited or saved.

4.14.1 Node Mapping

It is often desirable to use textual node names instead of the integers assigned by *Xic* as node references in SPICE output. This at times produces more readable SPICE output, and may be essential if the output is processed by other applications. A problem is that *Xic* generates the default node numbering internally, thus it may vary unpredictably if the circuit is modified. By using node names supplied by the user or based on the strings of attached vcc devices and terminal names, the node names become predictable and repeatable after circuit modification.

Node mapping is enabled by activating the **nodmp** button in the electrical mode side menu. When the **nodmp** button is on, node mapping will be performed when SPICE output is created. A property is added to the current cell, so the initial state of the **nodmp** button represents the state under which the cell was saved.

Pressing the **nodmp** button, when inactive, brings up the **Node Mapping** panel. If the button is active, one should turn the button off, then on again to bring up the panel. The panel has four buttons: **Dismiss**, **Rename**, **Remove**, and **Point**. The **Dismiss** button retires the panel, without changing the state of the **nodmp** button. The **Rename** button allows a name to be given to a selected node. The **Remove** button will delete the name assignment of the selected node, if one was given previously with the **Rename** button. The **Point** button, when active, allows nodes to be selected in the panel display by clicking on nodes in the drawing window.

The panel contains two text areas, with the left area listing the nodes, and the right area listing the device and subcircuit terminals connected to the selected node. Nodes can be selected by clicking on the text in the left text window, or by clicking in the circuit if the **Point** button is active.

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 when node mapping is active. If the second column entry is blank, the internal node number will be used.

The entry in the second column will be blank unless:

1. The node has been assigned a name with the **Rename** button.
2. The node is connected to a vcc type of device.
3. The node is connected to one of the cell terminals.

The precedence is in the order listed. If an entry exists in the second column, that token will indicate the node. This will not change if the circuit changes.

Node mapping is performed only when the **nodmp** button is active. When the button is inactive, the internal number will be used for all nodes.

The node referencing works by association with a device terminal. 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.

The names used for device terminals are the terminal names as supplied in the node properties in the device library file. However, only "physical" terminals will have an assigned name. Device contacts such as the phase node of a Josephson junction, branch nodes, and nodes of current and voltage sources have no implementation in a physical layout and have no names. In the listing in the right-hand panel, the terminal name for these "unnamed" terminals 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 or subcircuit (subcircuits may have non-implemented contacts as well).

When a node is selected, a small box is shown over all associated connections in the electrical view.

4.15 The plot Button: Generate SPICE Plot

The **plot** button, available only in electrical mode, gathers input for a `.plot` line for inclusion in a SPICE deck generated with the **deck** command, or for immediate display after a SPICE run (with the **run** command) if the *WRspice* program is available.

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 the phase node of a Josephson junction. By convention, these are indicated with a '+' mark in the symbol. For most 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.

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.

The **Enter** key terminates entry, and creates the plot if *WRspice* is available. The command string can be edited in the usual way, thus functions of the variables, for example, can be plotted (but this

syntax only applies to *WRspice*). In the **deck** command, the string will be added to the SPICE listing, when generated, as a `.plot` line.

4.16 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. When A polygon is created by clicking on each vertex in sequence, and is terminated by clicking again on the initial vertex, or clicking on the final vertex twice. The vertices can be undone and redone with the **Tab** key and **Shift-Tab** combination, which are equivalent to the **Undo** and **Redo** commands.

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 disabled by setting the NoSnapToEdges variable.

When adding vertices during polygon creation, the angle of each segment can be constrained to a multiple of 45 degrees with the **Constrain 45** button in the **Attributes 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.

Constrain 45 not set		
	Shift up	**Shift** pressed
Ctrl up	no45	reg45
Ctrl pressed	simp45	simp45

Constrain 45 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, polygons have no electrical significance, but can be used for illustrative purposes.

4.16.1 Polygon Vertex Editing

If a polygon is selected before entering the **polyg** command, a vertex editing mode is active on the selected polygons. Each vertex of the selected object is marked with a small highlighting box. Clicking on a vertex will delete the vertex. Clicking on the edge of a selected polygon away from an existing vertex will create a new vertex, which can subsequently be moved. To move vertices, one or more must

be selected. This 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, all vertex marks disappear, except for the selected vertices, which are marked with a slightly larger 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.

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 delete 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 45** button in the **Attributes Menu** is set, **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.

4.16.2 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).

4.17 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.

4.18 The round Button: Create Disk Object

The **round** button, only available in physical mode, will create a disk 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.

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 disabled by setting the NoSnapToEdges variable.

If the SpotSize variable is set to a value (with the **!set** command), the figure is constructed somewhat differently. With the SpotSize variable set to 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, 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. This conditioning, with SpotSize set nonzero, applies only to objects created with the **round** and **donut** commands, and not the **arc** command or general polygons.

4.19 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 SPICE analysis string to run. This should be in a format understandable to *WRspice* as an interactive-mode command. The run is then performed.

The link is established to the SPICE server 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, and *WRspice* is running on the local machine. 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.

An asynchronous *WRspice* interface is now the default for all operating systems supported with binary releases of *Xic*. 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.

If the variable NoAsyncSpice is set (with the **!set** command), the synchronous *WRspice* interface is used. In this case, there is no pop-up, and status is indicated in the prompt area. Pressing **Ctrl-C** over the controlling terminal window or over a drawing window pauses the analysis. The user should never press command buttons during an analysis if the interface is not asynchronous, as this is inviting a program crash. With the asynchronous interface, it is safe to continue working in *Xic* while the analysis is running.

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.

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.

To access the file data for plotting in *Xic*, press the **run** button, and enter for the analysis string "`load` *filename*", where *filename* is the name of an existing rawfile. The **plot** command can then be used to access the vectors as usual. When the analysis string is of the form "`load` *filename*", then no analysis is performed, but rather the *filename* (which must be a pre-existing rawfile) is loaded into *WRspice*. The **plot** command can then be used as usual to view the data in *filename*, if the plot data came from a previous run of the current circuit (there is no checking for consistency). It is not an error to load an inconsistent rawfile, and in fact the **plot** command can be used, but vector names must be entered from the keyboard.

4.20 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.

Boxes created on the SCED layer will be shown with the fill attributes assigned to that layer (solid fill by default). Boxes created on other layers will be shown as empty fill with a dashed border. If a fill pattern or solid border is desired on these layers, a polygon should be used.

Polygons are always shown with the current layer's fill attribute.

Arcs are actually wires, and as such should not be used on the SCED layer. Although there is no error, arc vertices are considered in the connectivity establishment, thus adding inefficiency. If the current layer is SCED when an arc is created, the arc is instead created on the next layer. If the arc must be on the SCED layer, the **Chg Layer** command command in the **Edit 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

4.21 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 maximum number of sides can be set with the MaxRoundSides variable. If not set, a reasonable default is used.

In electrical mode, this function is available in the menu presented by the **shapes** button.

4.22 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 45** button in the **Attributes Menu** is active, the angle will be constrained to multiples of 45 degrees. Pressing the **Shift** key will remove the constraint. If the **Constrain 45** button is not active, holding the **Shift** key will impose the constraint. Thus the **Shift** key inverts the effect of the **Constrain 45** button. However, if the selected objects include a subcell or label, the angle will always be constrained to multiples of 45 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, 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.

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 will be placed on the new layer. Subcells are not affected. If in layer-specific mode, only objects whose layer was the original current layer will be changed to the new layer. If not in layer-specific mode, all new objects will be placed on the new layer, no matter what their original 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.

4.23 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, on the current layer in layer-specific mode, or any wires if not in layer-specific mode, *Xic* prompts for a new wire width for the selected wires. In layer-specific mode, only wires on the current layer have their widths changed, otherwise selected wires from all layers 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.

4.24 The subct Button: Set Subcircuit Connections

The **subct** button, only available in electrical mode, allows electrical connection points to be added to a circuit. This is necessary if the circuit is to be used as a subcircuit in another circuit. The terminals are points at which electrical connections are defined, as in the SPICE subcircuit definition. These points are made visible with the **terms** button. While in physical mode, the **Show Terminals** button in the **Extract Menu** will also enable the display of terminals in electrical mode subwindows.

Terminals can normally be located only at connection points of the underlying geometry. Clicking on an existing terminal will delete it, unless the **Shift** key is held while clicking, in which case the terminal can be dragged to a new location (also over a connection point).

When a new terminal is created, or if an existing terminal is double-clicked on with the **Shift** key held (i.e., moved nowhere), the user is prompted for a name for the terminal. Pressing **Esc** or **Enter** or clicking button 1 anywhere in the drawing window will accept the 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. 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).

To change the name of an existing terminal, press **Shift** and double click on the terminal (no need to press **Shift** if in symbolic mode). In physical mode, terminals can also be renamed with with the **Edit Terminals** command in the **Extract Menu**.

In symbolic mode, i.e., the **symbl** button is active, terminals can not be added or deleted, however they can be moved to new locations consistent with the symbolic representation. There is no need to hold the **Shift** key in this case. The terminal marks can be moved anywhere in symbolic mode, as the "real" location remains fixed at the point set in normal mode.

4.24.1 Virtual Terminals

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. Virtual terminal internal node numbering is sequential starting at 1, with conventional nodes, if any, assigned higher values.

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.

4.25 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.

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, and it is not possible to set the display mode on a per-instance basis.

4.26 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 **Show Terminals** command in the **Extract Menu** was active. 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.

4.27 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, and is terminated by clicking on the final vertex twice. 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 **dots** button can be used to display connections. The vertices can be edited to remove or reestablish connections.

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 disabled by setting the NoSnapToEdges variable.

When adding vertices during wire creation, the angle of each segment can be constrained to a multiple of 45 degrees with the **Constrain 45** button in the **Attributes 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.

Constrain 45 not set		
	Shift up	**Shift** pressed
Ctrl up	no45	reg45
Ctrl pressed	simp45	simp45

Constrain 45 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.

Existing wires can be converted to polygons through selection and execution of the **polyg** command.

4.27.1 Wire Vertex Editor

If a wire is selected before entering the **wire** command, a vertex editing mode is active on the selected wires. Each vertex of the selected object is marked with a small highlighting box. Clicking on a vertex will delete the vertex. Clicking on a selected wire away from an existing vertex will create a new vertex, which can subsequently be moved. To move vertices, one or more must be selected. This 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, all vertex marks disappear, except for the selected vertices, which are marked with a slightly larger 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.

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 delete 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 45** button in the **Attributes Menu** is set, **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.

4.28 The xor Button: XOR 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 disabled by setting the NoSnapToEdges variable.

This page intentionally left blank.

Chapter 5

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.

File Menu			
Label	**Name**	**Pop-up**	**Function**
New	`new`	none	Open temporary cell
Open	`open`	**File Selection**	Open file
Save	`sv`	none	Save file
Save As	`save`	none	Save file, rename
Print	`hcopy`	**Print Control Panel**	Hard copy plot
Files List	`files`	**Files Listing**	List search path files
Libraries List	`libs`	**Libraries**	List libraries
Cells List	`cells`	**Cells Listing**	List cells in memory
Show Tree	`tree`	**Cell Hierarchy**	Display cell hierarchy
Quit	`quit`	none	Exit *Xic*

5.1 The New Button: Open Temporary Cell

The **New** button 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.

5.2 The Open Button: Pop Up File Selection Panel

This button 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.

5.2.1 The File Selection Panel

The **File Selection** panel allows the user the 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.

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.

In the Unix/Linux version, the '+' box will appear with subdirectories only after the subdirectory is selected. In the Windows version, the the '+' boxes will be visible without selection.

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.

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.

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 actions are slightly different for Unix and Windows systems. In Unix/Linux, the normal operation (button 1 with no keys pressed) for drag/drop is copying. The other options are as follows:

Unix 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.

Under Windows, only button 1 can be used for dragging, and there is no "Ask" option. The basic operation (no keys pressed) is `move`, however this can be reset by the underlying window. In Windows, directories can only be moved, not copied.

Windows Operations	
Button 1	Copy
Shift-Button 1	Move
Control-Button 1	Copy

Only one file or directory can be selected. When the operation is `copy`, the cursor icon contains a '+' in all cases, which will change when the **Shift** key is pressed (if a `move` operation is permitted) indicating the change to 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 **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 from an **Open** or **Open File** menu button, but is not shown when the pop-up appears 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.

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.

5.3 The Save Button: Save Cell

The **Save** button allows saving unsaved work to disk files, under the present file/cell name.

If editing a cell from the device library, the **Save** command will bring up the **Library Cell Parameters** panel (see A.5.1), 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, if there are cells in memory that have been modified, a pop-up will appear. 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 will set the states of all these words: **Save All** sets them to "`yes`", **Skip All** sets them to "`no`". The third column gives the type of file that will be created or updated: '`X`' for *Xic* native, '`B`' for CGX, '`C`' for CIF, '`G`' for GDSII, and '`O`' for OASIS. This entry is shown in color, and the color used for archives is different than the color used for native cell files. The fourth column is the full path name of the file that will be written if the second column is "`yes`". While the pop-up is visible, all 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.

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** command 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).

The **!sa** command has the same effect as invoking **Save**. The same pop-up appears at other times when the user should make a decision about saving work, such as when exiting *Xic*.

5.4 The Save As Button: Save Cell, Renaming

The **Save As** button 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 A.5.1), 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.

A format change can be coerced by supplying an extension to the name, or, if the path given is a path to an existing directory, the entire cell hierarchy can be saved as native cell files in that directory. In particular, giving "`.`" allows cell files to be written in the current directory. The following extensions are recognized:

CGX	"`.cgx`"
CIF	"`.cif`"
GDSII	"`.gds`", "`.str`", "`.strm`", "`.stream`"
OASIS	"`.oas`"
Native	"`.xic`"

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.

The archive formats always save the full hierarchy. Giving the "`.xic`" suffix will save the current cell only, under a new name including the suffix. Giving a directory path allows saving the entire cell hierarchy as native cell files, without name change.

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.

5.5 The Print Button: Print Control Panel

The **Print** command 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.

To improve visibility of the plots, the layer fill patterns, colors, and other attributes are set separately and can differ from those normally displayed on-screen. These alternate colors, fill patterns, grid display, and layer visibility choices are in effect only when the **Print** command is active. The **Set Color** and **Set Fill** commands in the **Attributes Menu** work identically to the normal commands, however the alternate colors and fill patterns are set. The **Set Grid** command in the **Attributes Menu** allows modification of the grid style displayed in the plot. Layers can be made invisible (or visible) during plotting with button 2 selections in the layer menu. Changes made to these attributes can be saved to a new technology file with the **Save Tech** command in the **Attributes menu**.

It is possible to define layer presentation attributes specific to individual hardcopy drivers, and to suppress the use of the "Alt" attributes for individual drivers. This can be enabled by setting various parameters in the device driver blocks near the end of the technology file.

When in hard copy mode with a driver containing layer specifications, and one changes the fill pattern, color, or visibility status of a layer specified in the driver block, the layer specification in the driver block will be updated, and not the "Alt" layer.

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 (several tens of megabytes may be needed for a color Versatec plot) the XIC_TMP_DIR environment variable should be set to a path to a suitable directory.

5.5.1 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.

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`" 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. A shell script `wr_lp` is provided for this purpose. Some operating systems provide a "`-r`" option to the "`lpr`" command which will remove the file when printing is complete. This option can be used in the command string as an alternative to using the wr_lp script. Also, large plots may require the use of the "`-s`" option to lpr, which will avoid a system limit on file size.

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 with the **Legend** button. 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 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 **Portrait** and **Landscape** buttons. These interlocking buttons switch between portrait and landscape orientation. 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** button is active, 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.

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.

5.5.2 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 two-color PostScript graphics list representing the plotted area.

PostScript line draw, color
: This produces an RGB color PostScript graphics list representing the plotted area.

Versatec, mono
: This produces monochrome output in a format suitable for Versatec electrostatic plotters and equivalent.

Versatec, color
: This driver produces output for a color Versatec electrostatic plotter or equivalent.

HP laser PCL
: This driver produces 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.1.6) for more information.

WordPerfect Graphics, mono
: This driver produces two-color output in the WordPerfect 5.1 graphical format, for inclusion in documents. In *Xic*, the fill patterns are defined in the technology file with the `WpFilled` keyword. Other fill pattern definitions are ignored. See the description of the WpFilled keyword in the technology file (section A.1.6) for more information.

WordPerfect Graphics, color
: This driver produces full-color output in the WordPerfect 5.1 graphical format, for inclusion in documents. In *Xic*, the fill patterns are defined in the technology file with the `WpFilled` keyword. Other fill pattern definitions are ignored. See the description of the WpFilled keyword in the technology file (section A.1.6) 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 source, 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.1.6) for more information.

Imlib: 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

On Unix/Linux systems, if you have the **ImageMagick** or **netpbm** packages installed (available as freeware on the Internet) then many more formats are available, including GIF. The `imsave` library, which is used to implement this driver and otherwise generate image files, uses 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/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, which will only be visible if no frame is used. 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.

Versatec plots may require large temporary files, several tens of megabytes for color. These are processed internally in segments, so generation is fairly efficient. Color PostScript bitmaps can also grow quite large, thus it is recommended that the compressed format be used. These bitmaps are processed entirely in virtual memory, thus systems with limited core may require a bit of time to produce larger bitmaps.

5.5.3 `wr_lp` script

The default print command for many of the hard copy drivers is a call to the script "`wr_lp`". This script is located in the same directory as the program binaries, and can be customized by the (super) user. The script is a wrapper around the system printer command (usually "`lpr`") and is designed to solve a problem. The application will not delete temporary files created for printing/plotting, since there is no way generally for the application to know when the job is finished and it is safe to delete the file. The `wr_lp` script runs the Unix "`at`" command, which will delete the file one hour after the print job is submitted. This is true whether or not the application is running, or if the user is no longer logged in. See the manual page for the `at` command to make sure that it is set up correctly.

Some versions of the `lpr` command provide a "`-r`" option, which will remove the file when printing

is complete (see "`man lpr`"). Use of this option is an alternative to using the `wr_lp` script.

5.6 The Files List Button: File Listing Panel

This selection will bring up the **Files Listing** panel which lists the symbol files found along the search path. The panel can be used to open files for editing and placing, among other useful features.

Under Unix/Linux, The main window displays a "notebook" listing of the symbol files found along the search path. Each directory in the search path has a separate page in the notebook listing. Clicking on the notebook tabs will bring the pages into view.

Under Windows, the pop-up contains a drop-down menu which has an entry for each directory in the search path. 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, looking for changes. Changes found are reflected in the listings, so that changes to the directory, perhaps caused by some other program, are shown. This is not possible 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.

A file can be selected by clicking on the name, and while selected it will be highlighted. Selected file names are acted on by the **Edit**, **Master**, and **Contents** command buttons.

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. These in turn can be selected for use as input by the **Edit** and **Master** functions to be described. The panel will not appear until a file has been selected. When new file selections are made, the content listing is updated.

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. 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.

The **Edit** and **Master** buttons allow cells from the selected files to be opened for editing or placement, respectively. If the **Contents** button is inactive, if a file name has been selected in the listing, pressing the **Edit** button will make the top cell in the file the current editing cell, just as if the name had been read in with the **Edit** command in the **Edit Menu**. If the current cell is modified, the user will be given the opportunity to save it before switching to the new cell. The cell name is deselected. If no file name is selected in the list, pressing **Edit** enters edit mode. In edit mode, clicking on a file name will cause that file to be opened for editing, i.e., the top level cell becomes the current editing cell. If the selected file is a library, a content window will appear, allowing the user to choose a reference from the library.

With the **Contents** button active, a selection must be made in the content listing window if the selected file is an archive file, the top-level cell is not automatically edited in this case. Similarly, no action is taken on a library file unless a reference has been selected in the content window. Instead, selecting a cell from the contents list will extract that cell and its descendents from the archive and open it for editing or placement, depending on the states of the **Edit** and **Master** buttons. The behavior for other types of files is unchanged.

The **Master** button uses the same selection logic, however the selected cell will 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 this panel, then clicking on locations in a drawing window.

5.7 The Libraries List Button: List Open Libraries

This command produces a pop-up which displays a listing of libraries that are currently "open". An open library is searched when a cell name is being resolved. The open libraries are searched before the files in the search path. A library is opened if it is ever listed in a content window from the **Files Listing** panel, or if a cell from that library is ever opened, such as by giving "*/path/library cellname*" to the **Edit** command in the **Edit Menu**, or if opened with the **Open** button (see below).

Libraries are listed and searched in the order opened. When resolving a reference, the first match will supersede any later entries. Clicking on a library in the listing will display a listing of references in that library.

There are five buttons: **Edit**, **Master**, **Open**, **Close**, and **Help**. If **Edit** or **Master** is active (at most one can be active at once), clicking on a cell reference in a content window will generally edit that cell or add that cell to the master list in the **Cell Placement Control** pop-up. This will be 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 **Open** button presents a list of library files found in the 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 open libraries have an "O" in the first column. Clicking on the entries in this list will open/close the libraries. When open, the library will also appear in the main pop-up list.

The **Close** button will close the library that is currently selected in the main list, after confirmation. Since libraries in this list may have been opened through a lengthly indirection chain, a confirmation is required to actually close the library.

With the **Edit** or **Master** button active, clicking on a reference to another library that does not have a *cellname* will pop-up a content window for that library (opening it in the process).

5.8 The Cells List Button: Cell Listing Panel

The **Cells List** button is used to bring up the **Cells Listing** panel, listing the cells presently in memory, in the current symbol table. The panel can be used to select cells for editing or placing, among other useful features.

The listing is a drag source — cell names can be dragged and dropped into drawing windows, to display or edit that cell.

The main window displays a listing of cells presently in memory. 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. A cell name can be selected or deselected by clicking on the name. Only one name can be selected at once, and it will be highlighted. Selected names are acted on by other buttons of the panel.

The cell name taken as input by the various commands in this panel can be supplied as follows. Highest precedence is given to a cell name selected in the listing by clicking on the name. If no such selection exists, and text is selected in the window brought up by the **Info** button in the **Cells Listing** panel or the **Info** button in the **View menu**, that text, if it is a known cell name, will be taken as input. The intent of this feature is to provide easy access to the cells listed as parents or subcells in the listing for a cell. If no such selection exists, and the **Cell Hierarchy** pop-up is visible and has a selection, then

that selection will be used. This pop-up is created by the **Show Tree** button in **File Menu**, and the **Tree** button in the **Cells Listing** pop-up. If there is yet no selection, and a cell is selected in one of the drawing windows, then that cell name is used, except for use with the **Edit** button.

The **Clear** button in the **Cells Listing** panel 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 database 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.

The **Tree** button is used to bring up the **Cell Hierarchy** pop-up, which can also be initiated with the **Show Tree** button in the **File Menu**. If a cell name has been selected in the cell name listing, the **Cell Hierarchy** pop-up (it might already be visible), or a cell is selected in a drawing window before the **Tree** button is pressed, the hierarchy of the cell will be displayed in the **Cell Hierarchy** pop-up.

If no cell name has been selected, the tree mode becomes active when **Tree** is pressed, so that selecting a cell in the **Cells Listing** will cause that cell's hierarchy to displayed in the **Cell Hierarchy** pop-up.

The **Edit** button is used to select a cell for editing. If a cell name has been selected in the listing or the **Cell Hierarchy** pop-up, pressing the **Edit** button will make that cell the current editing cell, just as if it had been read in with the **Edit** command in the **Edit Menu**. If the current cell is modified, the user will be given the opportunity to save it before switching to the new cell. The cell name is deselected.

If no cell is selected in the list or the **Cell Hierarchy** pop-up, pressing **Edit** on the panel invokes edit mode. In edit mode, clicking on a cell name in the panel will cause that cell to become the current editing cell.

The **Master** button allows a cell to be selected for placement. If a cell name has been selected in the listing, the **Cell Hierarchy** pop-up, or in a drawing window, pressing the **Master** 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.

If no cell has been selected, master mode becomes active, in which case clicking on a cell name causes it to become the current master. Master mode is retained until one of the other buttons is pressed, or **Master** pressed again.

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. If there are no selections in the cell listing, the **Cell Hierarchy** pop-up, or in the drawing window, the copy mode becomes active. Selecting a cell in the cells listing will cause a dialog box to appear asking for a new name for the cell. If a selection is active, pressing the **Copy** button will immediately bring up the dialog box. In either case, a copy will be made if the user enters a valid new name. The new name will become highlighted in the cell listing.

After the copy function in the **Cells Listing** panel is invoked, if any cells are selected in a drawing window, the user is prompted by a pop-up dialog whether to replace the selected cells with the copy. Thus, to make a custom instance, one selects an instance, uses **Copy** with replace, then pushes to the new instance and performs the changes, using the **Push** command in the **Edit Menu**. In electrical

mode, if a name was assigned to the original cell, the new cell retains that name. The replace dialog appears even if no copy is made, if the user does not change the cell name in the copy dialog and just presses **Enter** or the **Dismiss** button. 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 **Edit 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.

The **Rename** button allows a cell in memory to be given a new name. All references to the cell throughout the database 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.

If a cell name is selected in the list, the **Cell Hierarchy** pop-up, or a cell is selected in a drawing window, pressing the **Rename** button brings up a dialog box asking for the new name. If no cell is selected, pressing the **Rename** button enters a mode where selecting a cell in the cells listing causes the dialog to appear. In either case, the renaming is effective if a valid new name is given. 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).

The **Search** button enables a mode that lists the subcells contained in rectangular areas of the screen. In this mode, when the user clicks or drags in a drawing window, the subcells contained in the displayed hierarchy that overlap this area are listed in a pop-up window. Each cell that overlaps the click location or rectangle is listed, through the entire depth of the hierarchy. A cell name is listed only once, if it is instantiated multiple times.

The search results show the search area coordinates in microns, unless the InfoInternal variable is set with the **!set** command, in which case it uses internal units.

The **Info** button produces a pop-up that provides information about subcells and other objects. If a cell name has been selected in the listing, the **Cell Hierarchy** 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 are no selected cell names, the mode becomes active, so that clicking on a cell name in the listing will bring up an **Info** window describing the cell. A similar capability exists for the **Info** button in the **View Menu**.

The **Show** button enables a mode where cell instances are highlighted in the display. If a cell name has been selected in the listing, the **Cell Hierarchy** pop-up, or in a drawing window, before the **Show** button is pressed, 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.

If no cell name has been selected, the mode becomes active when **Show** is pressed, so that clicking on a cell name in the listing will cause that cell to be highlighted in the drawing windows. Only one cell can be highlighted at once.

5.9 The Show Tree Button: Show Cell Hierarchy

This button brings up a window containing a tree diagram of the hierarchy of the current cell. Each subcell is initially shown unexpanded, but these can be expanded by clicking on the "+" symbol. Clicking elsewhere in the line will select the particular subcell. These selections will be used by the commands in the **Cells Listing** panel if no selections are in force in that panel. Subcells can be unexpanded bu clicking on the "−" symbol that replaced the "+" symbol when the subcell was expanded.

The listing is a drag source — cell names can be dragged and dropped into drawing windows, to display or edit that cell.

5.10 The Quit Button: Exit *Xic*

Pressing this button 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.

This page intentionally left blank.

Chapter 6

The Edit Menu: Modify Geometry

The **Edit Menu** contains commands which alter the current design. Many of these commands have shortcuts, so an experienced user may not often use this menu.

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**
Edit	`edit`	none	Edit new cell
Current Transform	`xform`	**Current Transform**	Set transform
Place	`place`	**Cell Placement Control**	Place subcells
Push	`push`	none	Edit subcell
Pop	`pop`	none	Edit parent cell
Flatten	`flatn`	**Flatten Hierarchy**	Flatten hierarchy
Create Cell	`crsym`	none	Create new cell
Properties	`prpty`	**Property Editor**	Edit properties
Cell Properties	`cprop`	**Cell Property Editor**	Edit cell properties
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

6.1 Cell and Object Properties

A property consists of an integer and a string, and is stored in the symbol file. There are a number of properties that are recognized by *Xic*. Every database object, including cells, instances, and geometrical objects, has the native ability to accept properties, though this is enabled selectively. The **Edit Menu** contains the main commands for applying properties: the **Properties** command which brings up the **Property Editor** for manipulation properties of objects, and the **Cell Properties** command which

brings up a similar editor for manipulating properties of the current cell.

In schematic layouts, properties define electrical parameters for devices, and are used when generating SPICE output. In physical layout mode, almost any number/string pair can be added, though *Xic* reserves some numbers for internal use, such as to store the grid used for the layout, or the GDSII end style for wires. In addition, there are reserved "pseudo-property" numbers that are not saved as properties, but rather set or return information about the object.

6.1.1 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 can be applied to devices and subcircuit instances:

name	the device name given to SPICE
param	the device initial condition and other parameters for SPICE
other	anything, not used internally
nophys	always "`nophys`"

These properties can be applied to device instances only:

model	the name of a SPICE device model
value	the device value as given to SPICE

In addition, internally generated properties, such as node numbers, are listed in the display. There can be only one name, model, value, param, and nophys property, but arbitrarily many other properties per device. The name property will always be present. Assigning a name property means that the assigned name will be used in SPICE listings, otherwise *Xic* will generate a name for the device. Deleting a name property simply deletes the user's name, if any. The other properties are not used internally, and the text can be any string. They are often useful for storing alternate strings for the model, value, and param properties. The nophys property is a flag to indicate that the device or subcircuit has no physical representation in the layout, and will be ignored in LVS testing.

The device line in a SPICE file looks like:
(*name*) (*nodes ...*) (*value* or *model*) (*param*)

The *name* field is either the name property given, or an internally generated name. The *nodes* field is internally generated, but optionally passes through a node name mapping as described in 4.14.1. Only one of the value or model properties is used. If both are set, the model has precedence. This is followed by the param (initial condition and parameters) string, if present. The precise format of the strings for each of these properties is determined by the device being set, and the details of the version of SPICE being used. The first letter of a name property should be the character used to key the device type to SPICE. The user should be familiar with these requirements.

In addition, it is possible to apply the param property to subcircuit instances, and also to the current cell with the **Cell Properties** command. This provides support for subcircuit parameterization, which is available in *WRspice* and some other simulators.

Unlike all other properties, after changing a param property of a cell, using the **Cell Properties Editor** or otherwise, all instances of that cell are examined and the param property of the instances may be updated.

1. If an instance has no param property or the instance param property text is the same as the old

cell param property text, or there was no previous cell param property, the instance property is updated to be a copy of the new cell param property.

2. If a cell param property is deleted, param properties found in the instances with matching text will also be deleted.

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 resistor. Give the resistor a value property consisting of some word, say "`rshunt`". Using the **Cell Property 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. Note that "`rshunt=2.5`" appears in a label next to each instance. Select one of the labels, and using the label editor change the string to "`rshunt=1.25`". This will change the resistor value in that instance (only) to 1.25 ohms.

6.1.2 Physical Mode Pseudo-Properties

In physical mode, the listing of properties contains "pseudo-properties" which 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. There are a number of reserved property numbers which can be applied to physical objects (this feature applies in physical mode only). Setting the property will change a geometric or other attribute of the object. Reading the property will provide the value of the geometric or some other parameter. The pseudo-properties are listed below.

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 subcells. 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 "1.0" for objects other than subcells, and the magnification value for subcells. When applied to objects other than subcells, all coordinates of the object will be scaled by the given value. When applied to subcells, the instance magnification factor will take the new value. Note that the instance origin will not change, however the scaling given to other objects in general implies a translation as well as a change in size.

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 wires, and can be applied to wires. The property string is the end style code, which is an integer 0, 1, or 2. If 0, flush ends are used. If 1, the wire is extended by one half of the width, and the end is rounded. If 2, the wire is extended by one half of the width, and the end is Manhattan. 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 labels, and can be applied to labels. The property string is a hexadecimal integer representing a transformation code.

Bits	Description
0–1	0-no rotation, 1-90, 2-180, 3-270
2	mirror y after rotation
3	mirror x after rotation and mirror y
4	shift rotation to 45, 135, 225, 315
5–6	horiz justification 00,11 left, 01 center, 10 right
7–8	vert justification 00,11 bottom, 01 center, 10 top
9–10	font

When applied to a label, the label will rendered using the new code. 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.

The settable pseudo-properties for an object are listed in the Properties Editor, along with the "real" properties. These can be changed in the same way, which will produce physical changes to the object.

6.2 The Edit Button: Edit Cells

The **Edit** button is used to read a file and load a cell for editing. The button presents a drop-down menu containing the names of the last eight cells opened for editing, plus a "**new**" entry. Select **new** to open a new cell by giving a name on the prompt line. The default name 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 cell name.

- A selection in the **File Selection** pop-up from the **Open** button in the **File Menu**.
- A selection in the **Cells Listing** pop-up from the **Cells List** button in the **File 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 **File 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.

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**. This is a quick backdoor for instantiating cells recently edited.

The **Edit** button also allows text editing of two special files. 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, 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.

If a cell is opened for editing that contains empty symbols, the user is given the option of deleting these references. This test can be suppressed by setting the NoCheckEmpties variable. If empty symbols are found in the hierarchy, a pop-up appears, which allows the user to delete the empty symbols. This test can be performed at any time with the **!empties** command.

6.2.1 Input to the Edit Command

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 **Edit** buttons in the files, cells, and libraries listing pop-ups in the **File Menu**. These are all equivalent to opening the cell with the **Edit** command, so that the information in this section applies in those cases.

The name provided can be a file containing data in a supported archive format, 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 editing. 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.

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**.

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).

While an input 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 1.4).

When reading an archive 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.

In archive file conversions, an alias file (see 9.2) may be created or used. The alias file, which has the same name as the archive file with an extension "`.alias`", contains symbol name mappings between *Xic* and the archive file. If present, this file is used to translate symbol names when an archive file is read.

When reading in an archive file, various other cell name modifications (see 9.1) may be applied, such as case conversion. Some changes are automatic, such as name changes to avoid conflicts with device library cells. If a name is changed, a warning message is added to the conversion log file indicating the change, and the change may be reflected to the alias file, if enabled.

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 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 (9.5).

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.

The **Edit** command can access the Internet. The name given to the **Edit** command can be in the form of a URL, followed by a subset of the "`httpget`" options. The URL must begin with "`http://`" or "`ftp://`", and the file is expected to be suitable *Xic* input.

The options that can follow the URL are:

`-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.

`-u` *user*

`-p` *password*
: These are the http "basic" authentication user/password. Note that these are not the site login/password, which should be part of the URL if needed. Rather, they are for accessing restricted pages through an http server which supports basic authentication.

6.2.2 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 **!symtab** interface command. 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.

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.

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 pop-up comes in two different forms, depending upon the type of file being opened. For CIF files the "batch mode" **Merge Control** will appear. For each cell in the pop-up listing, one can select whether or not to overwrite the physical or electrical part of the cell in memory by clicking on the "yes" and/or "no" that follow each cell name. One can also set the values for all cells with the buttons. Pressing the **Apply** button will remove the pop-up and continue the conversion. Pressing **Abort** or dismissing the pop-up will abort the conversion.

Similarly, when reading in a non-CIF cell hierarchy with cells already in memory, a different **Merge Control** pop-up appears which 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** button in the **Conversion − Import** 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 **Conversion − Import** 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.

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.

6.2.3 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 **Conversion – Import** panel.

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.

Xic checks for identical, coincident objects when reading input files, and prints a warning message in the log file if such objects are found. This test can be turned off by setting the NoCheckCoinc variable, or from the **Conversion – Import** panel. If the **Merge boxes ...** button in the **Conversion – Import** panel is set, duplicate objects (not subcells) will be deleted and no warning is issued. In addition, box objects will be clipped and/or merged so that no two boxes on the same layer overlap.

6.3 The Current Transform Button: Current Transform Panel

The **Current Transform** button in the **Edit Menu** brings up a 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.

Reflect X
Add a reflection of the x-axis to the current transform. This will affect moved or copied objects, and newly placed instances.

Reflect Y
Add a reflection of the y-axis to the current transform. This will affect moved or copied objects, and newly placed instances.

Rotation 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 arrow keys cycles through the transformation angles, whether or not the **Current Transform** panel is visible. The right and up arrows increase the angle, left and down arrows decrease the angle.

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.

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.

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.

6.4 The Place Button: Cell Placement Control Panel

This command 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 in the **Edit Menu** is active (the two buttons show the same status), and the user is not being prompted for a cell name (the "master") to place, 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, remains active. 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.

From the **Edit** command in the **Edit 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. 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 the lower left corner of the untransformed bounding box, while in physical mode. 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 cell origin is the reference point. 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.

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. 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 **Open** button in the **File Menu**.
- A selection in the **Cells Listing** pop-up from the **Cells List** button in the **File 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 **File 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 place 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 place panel is sensitive as a drop receiver. If a file name is dragged over the place 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.

The number of masters saved in the pull-down menu can be specified with the variable MasterMenuLength, which defaults to 25. This may not be optimum for some screen resolutions, and it does no good if the menu extends off-screen. To set the length to 15, for example, one would type (outside of any command)

```
!set MasterMenuLength 15
```

This must be entered before the place panel is popped up for the first time. This command would logically be included in a startup script. The list used in the drop-down menu consists of the most recent masters specified, either with the **New** button, or through the **Master** button in the **Cells Listing** or **Files Listing** panels from the **File Menu**.

Two mutually exclusive buttons select whether the cell reference point is the cell's origin (**Origin** button), or the lower left corner of its bounding box (**Lower left** button). This is the point that is attached to the pointer, and corresponds to where the user clicks to place the cell. These buttons have no effect in electrical mode, as the reference point is always the device or subcircuit's reference terminal.

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.

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 **Origin/Lower Left** buttons in the place panel. 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. 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 **Edit Menu** is active.

The **Dismiss** button will retire the **Cell Placement Control** panel. The place mode, if active, is *not* terminated, and additional cells can be placed. Place mode is terminated by pressing the **Esc** key, or by pressing the **Place** button in the **Edit Menu** a second time.

6.5 The Push Button: Push Editing Context

Pressing the **Push** button will push the editing context 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** button in the **Attributes Menu**.

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.

6.6 The Pop Button: Pop Context

Pressing this button 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 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 **File Menu** is invoked.

6.7 The Flatten Button: Flatten Hierarchy

This button brings up a small 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 1, only geometry in the subcell is flattened, and sub-subcells are placed. If "`all`", the entire subcell hierarchy is flattened. There is also a button to skip box and wire merging and undo list processing operations for speed. This may be desirable for flattening complex cells, which may take a long time to process. In this mode, there is no "undo" capability, however.

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".

The symbolic data from flattened subcells are added to the symbolic part of the current cell, in electrical mode. This will only be visible if the current cell is in symbolic mode (see the **symbl** button).

6.8 The Create Cell Button: Create Symbol

This command 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 symbol 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 CrsymOverwrite variable is set, existing cells in memory can be overwritten (use this with care).

6.9 The Properties Button: Set Properties Panel

This button 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 used when building a SPICE deck. When the **Properties** button is 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 **Properties** button was pressed, or were clicked on with the **Properties** button 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.

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. The **desel** button in the side menu will deselect any selected objects, and clear the list.

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 **Property Editor** is visible, clicking on an object with the **Shift** or **Control** key pressed will bring up 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.

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 default colors are:

sienna	ordinary (user-modifiable) properties
dark blue	physical mode pseudo-properties
black	internal properties

The colors can be modified with the **!set** command.

In addition, the value of the electrical mode name property is shown in color when the property is set. This property always exists, thus it might 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.

6.9.1 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 4.11.3) 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 4.11.4) 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.

6.9.2 The Add Button: Add New property

In physical mode, the **Add** button 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 properties. The **Name**

button allows the modification of a name property. The name property specifies the device 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.

The **Model**, **Value** and **Param** buttons allow addition of a model, value or param property, respectively. Only one of 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. The param property is a catch-all for additional parameters found in the device line in SPICE, such as initial conditions or device geometrical factors.

The **NoPhys** button allows addition of a nophys property. This property does not affect SPICE output, but specifies that the device or subcircuit 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.

There can be at most one each of these properties. This is enforced by *Xic*, i.e., attempts to add a second property of a given type will cause replacement, not addition.

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.

6.9.3 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.

6.10 The Cell Properties Button: Edit Cell properties

This button 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 param property of electrical cells, and the properties that control template cells in physical mode.

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 electrical mode, only the **Param** and **Other** entries are active. 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 the section on properties (6.1.1). 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.

In physical mode, there are four entries active, allowing modification of physical cell properties. 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. The **Tmpl Script** and **Tmpl Params** entries are used when defining template cells. These set the Template Script property and Template Params property, respectively. The **Other** 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.

6.11 The Undo Button: Undo Operation

The **Undo** command 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.

6.12 The Redo Button: Redo Last Undo

The **Redo** button 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.

6.13 The Delete Button: Delete Objects

The **Delete** button may be used to delete the selected objects. This is redundant, as selected objects can be deleted by pressing the **Delete** key.

6.14 The Erase Under Button: Erase Under Objects

This command 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.

If in layer-specific mode, only the current layer is erased, others are untouched. Otherwise, all unselected geometry is affected.

6.15 The Move Button: Move Objects

This command 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.

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** again shifts to the upper left corner, etc. 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 will be placed on the new layer. Subcells are not affected. If in layer-specific mode, only objects whose layer was the original current layer will be changed to the new layer. If not in layer-specific mode, all new objects will be placed on the new layer, no matter what their original 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.

6.16 The Copy Button: Copy Objects

This command 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. 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.

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 the replication count is greater than one, the command exits after the operation.

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. If the replication count is one, multiple copies are made by simply clicking on additional destinations.

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** again shifts to the upper left corner, etc.

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 will be placed on the new layer. Subcells are not affected. If in layer-specific mode, only objects whose layer was the original current layer will be changed to the new layer. If not in layer-specific mode, all new objects will be placed on the new layer, no matter what their original 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".

Copy operations can be also performed through the command line interface with the **!co** command.

6.17 The Stretch Button: Stretch Objects

The **Stretch** command 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 45** button in the **Attributes Menu** is set, **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 45** button can be used to constrain the stretch angle as described above.

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.

6.18 The Chg Layer Button: Change Layer

The **Chg Layer** button 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.

Chapter 7

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`	subwindow	New drawing window
Peek	`peek`	none	Show layers in area
Cross Section	`csect`	subwindow	Show layers in cross-section
Rulers	`ruler`	none	Add transient gradations
Info	`info`	**Info**	Show cell/object parameters
Bandwidth	`bw`	none	Show rendering time
Allocation	`alloc`	**Memory Monitor**	Show memory statistics

7.1 The View Button: Select Cell View

The **View** button produces a drop-down menu of view choices. In a given window, the last five views are saved in a list. Additionally, up to five views can be saved by pressing **Ctrl-N**. These are assigned name 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**.

7.2 The Physical Button: Show Physical Mode

The **Physical** button, available only in electrical mode, changes the main *Xic* 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 subwindows, the **Physical** button changes the view only. Editing can not be performed in a subwindow whose mode is not that of the main window.

While in a **Push** (see 6.5), the symbol currently being edited remains the current symbol, 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.

7.3 The Electrical Button: Show Electrical Mode

This command 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 subwindows, the **Electrical** button changes the view only. Editing can not be performed in a subwindow whose mode is not that of the main window.

While in a **Push**, the symbol currently being edited remains the current symbol, 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.

7.4 The Expand Button: Expand Subcells

The **Expand** button allows the expansion of subcells in the window to be controlled. 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.

After pressing **Expand**, a 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 subwindow is created, it inherits the expansion status of the main window. The expansion depth entered applies only to that window.

7.4.1 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 subwindow. Setting the expansion to "none" clears all expansion in peek mode.

7.5 The Zoom Button: Zoom In/Out

After pressing the **Zoom** button the user is prompted for a scaling factor, by which the present window width is multiplied. 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.

7.6 The Viewport Button: Create Subwindow

The **Viewport** button brings up a subwindow. The user is requested to point at the diagonal endpoints of the region to be displayed in the subwindow. Each viewport contains a menu of attribute buttons which apply to that window only. In particular, the subwindow can display cells in either electrical or physical mode, however editing operations are only possible if the subwindow mode and cell match those of the main window.

The subwindow has a set of menus which control attributes which can be set on a per-window basis. When the cursor is in a subwindow, characters entered are delivered to that subwindow, and an unambiguous sequence match will select a subwindow button. Matches are looked for in the subwindow menu, the main menu, and any pop-up menus, in that order.

The menu button commands are analogous to the commands found in the main menu. An exception is the **Dump To File** command, which allows the contents of the subwindow to be dumped to a disk file. When pressed, a file name will be solicited, and the contents of the subwindow 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` library, which is also used in the imlib print driver. See 5.5.2 for additional information.

Another exception is the **Show Location** command, which draws an indicator in the main window showing the display area of the subwindow. When this menu button is active, and both the subwindow

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 subwindow. This indicates the position of the subwindow display, assuming that the subwindow is showing a zoomed-in part of the display in the main window.

If a cell shown in a subwindow is the cell shown in the main window, with the same mode (physical or electrical), then all editing operations will work in the subwindow as well as the main window. The subwindow will display all highlighting, terminals, and other special markings. If the subwindow 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 subwindow, the subwindow will initially show the same cell as the main window.

There are presently two ways to set the cell displayed in a subwindow: the **!cc** command, and by drag/drop. In addition, the **!sw** command will swap the displays between the main window and one of the subwindows. The subwindows 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 subwindow, which will cause that cell/file to be opened and displayed in the window.

7.7 The Peek Button: Show Layer Composition

This command asks the user to define a rectangular area, 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 layer), can be reset with the PeekSleepMsec variable, which can be set to the delay time in milliseconds.

7.8 The Cross Section Button: Show Cross Section

This command 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, a subwindow 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 **Constrain 45** in the **Attributes Menu** is active, the angle is constrained to multiples of 45 degrees. If not set, 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 initial (anchor) point while in the **Cross Section** command.

If a layer has the DarkField attribute given, the cross section will show the inverse of the layer as it appears ordinarily. Thus a via, which might appear as a colored square in the main window, will appear as a hole in cross section, since the Via keyword implies DarkField. The DarkField keyword should be applied to other layers that print in reverse polarity for the cross section to be accurate (It is implied by GroundPlaneClear, as with Via). This can be done with the **Extraction Parameter Editor** from the **Edit Extraction** command in the **Extract Menu**, or by adding the DarkField keyword to the appropriate layer block in the technology file. If the Symbolic keyword appears, the layer will not appear in the cross section view.

The height of each layer, as drawn, is obtained from the Thickness parameter from the layer blocks in the technology file. If not given, a default thickness is assumed. The variable XSectThickness, which sets the default thickness, can be set with the **!set** command. This is an integer 100 – 100000 which represents the thickness in internal units (1000 units/micron). If not set, a value of 500 (0.5 microns) is used. The individual layer thicknesses are specified in microns, and can be added by editing the technology file or using the **Extraction Parameter Editor**. The thicknesses are scaled proportionally so the layers will always be visible. They appear with actual thickness when the viewing area is sufficiently small.

7.9 The Rulers Button: Create Rulers

This button 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 in physical mode, for example 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 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.

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. 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 **Constrain 45** button in the **Attributes Menu** is active, the angle is constrained to multiples of 45 degrees. If not set, 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, unless the RulerSnapToGrid variable is set. The period ('.') key toggles snapping to grid points while in the **Ruler** command.

7.10 The Info Button: Display Information About Objects

The **Info** button brings up a window which displays information about selected objects, or alternatively allows information about selected objects to be dumped to a file. When the command is entered, if there are selected objects, the user is asked 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 '**n**' is given at the first prompt, any selected objects become deselected. The user is prompted to click on objects, which will become selected. Only one object can be selected per pointer operation.

After an object is selected, a text box appears which provides information about the object, such as the object type, bounding box, coordinates, etc.

If the **Shift** key is held while the user clicks anywhere in a drawing window, information about the current cell is provided.

By default, dimensions are given in microns. If the variable InfoInternal is set (with the **!set** command) then dimensions are given in internal units (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.

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 **File 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.

7.11 The Bandwidth Button: Show Rendering Time

When the **Bandwidth** button is active, *Xic* will show the number of features drawn per second after each redisplay. Statistics will also be shown after design rule checking with the **Check In Foreground** command. These statistics may be useful for benchmarking purposes.

7.12 The Allocation Button: Show Memory Allocation

Pressing this button 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.

This page intentionally left blank.

Chapter 8

The Attributes Menu: Set Display Attributes

The **Attributes Menu** contains commands primarily for modifying the presentation format of *Xic*. Each subwindow 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	Set keyboard macro
Freeze Display	`freez`	none	Suppress redisplay
Constrain 45	`ang45`	none	Constrain angles
Show Context	`cntxt`	none	Show context in subedit
Show 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
Show Origin Mark	`mark`	none	Show selected physical instance origin
Tiny Boxes	`tinyb`	none	Outline tiny instances
Set Font	`font`	**Font Selection**	Set text-area font
Set Grid	`grid`	**Grid Parameters**	Set grid parameters
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

8.1 The Save Tech Button: Update Technology File

The **Save Tech** button will write an updated technology file in the current directory. This file provides setup information to *Xic*. The file will reflect the current settings of the configurable attributes.

8.2 The Key Map Button: Assign Keys and Macros

The **Key Map** command has two functions: mapping of needed function keys that might not be present on some keyboards, and mapping of keypress combinations to general key/button sequences (macros). The user is first prompted for which mode to enter.

8.2.1 Key Mapping

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 a 'y' response to the first prompt in the **Key Map** command. In this mode, 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.`*hostname*, where *hostname* is the name of the computer whose keyboard is being mapped. This file should be moved to the user's home directory or the system startup directory to create a permanent new keyboard mapping. This file will be read when *Xic* starts in future sessions.

8.2.2 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.

8.2.3 Keyboard Macros

Entering 'n' or just **Enter** at the first prompt of the **Kay Map** command 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 taken as the value of the HOME environment variable, if set.

Macros can be attached to any key combination of a keyboard key, except the **Enter**, **Esc**, and the "bare" modifier keys (**Shift**, **Ctrl**, etc). Under Unix/Linux, the macro supersedes any other function for the key, such as for menu accelerators, or other predefined operations in *Xic*, thus the key to be mapped should be chosen with care. Under Windows, The expansion of keyboard accelerators occurs before

macro expansion, so menu accelerator keys can not be used as macros. Also, the **Alt** key is treated specially by Windows, and therefor can not be used as part of the macro sequence.

Macros are *not* expanded when the input editor is active, i.e., the word "INPUT" is printed in red to the left of the prompt line. 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 ‘d’ is mapped to something, but you want to enter a literal ‘d’ to trigger the **Edit** command ("edi"). One would type "e‘di", after which the keypress buffer would contain "edi" triggering the **Edit** 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. **Bsp** removes the last event when defining a macro. To enter **Esc**, **Bsp**, or **Enter** into the macro, use **Ctrl-Esc**, **Ctrl-Bsp**, **Ctrl-Enter**.

For example, Suppose we want to define **Ctrl-X** to "press" the **Expand** button. In the **Key Map** command, press **Enter** at the first prompt (don't want the keyboard fix), and press **Ctrl** and **x** and release. In response to the next prompt, select the **Expand** 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. 14 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.

8.2.4 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 13.5) 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 1.2.6). 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*, use the **!logfiles** command, 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.

8.3 The Freeze Display Button: Suppress Redisplay

When pressed, 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 subwindows have an additional feature: frozen subwindows display the viewport of the main window, the reverse of the **Show Location** function for subwindows (which displays subwindow 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 to following procedure. Freeze the main window and read in the design. Bring up a subwindow by clicking twice outside of the cell boundary, so that the subwindow is empty. Freeze the subwindow, then press the **Home** key with the cursor in the subwindow 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 subwindow 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 subwindow.

8.4 The Constrain 45 Button: Constrain Angles

When on, 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 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** button is not active, holding **Shift** will impose the 45 degree angle constraint. Thus, the **Shift** key inverts the effective state of the **Constrain 45** button in these commands

8.5 The Show Context Button: Show Context in Subedit

With the **Show Context** button active, the context is displayed during a subedit, as initiated with the **Push** button in the **Edit Menu**. The context is the surrounding geometry in cells other than the cell being edited.

8.6 The Show Properties Button: Show Physical Properties

The **Show Properties** button enables the display of assigned properties on-screen while in physical mode. Properties can be assigned with the **Properties** command in the **Edit Menu**. 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.

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 numbered 7000-7104 are not displayed. These numbers are reserved for internal use should not be assigned.

If the variable EraseBehindProps is set (with the **!set** command), then the background behind the property strings is erased. This improves visibility of the strings.

8.7 The Show Labels Button: Display Labels

When the **Show Labels** button 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.

8.8 The Label True Orient Button: Display Labels Transformed

This button 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.

8.9 The Show Cell Names Button: Show Cell Names

When on, 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.

8.10 The Cell Names True Orient Button: Show Cell Names Transformed

This button 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.

8.11 The Show Origin Mark Button: Mark Subcell Origins

When the **Show Origin Mark** button is active, instances will have their origin points marked when selected. This applies to physical mode only.

8.12 The Tiny Boxes Button: No Expansion of Tiny Subcells

When this button 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 size threshold can be changed with the CellThreshold variable, which can be set with the **!set** command.

8.13 The Set Font Button: Set Window Fonts

This button brings up a 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.

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 (Unix/Linux only)
: 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 (Unix/Linux only)
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.

8.14 The Set Grid Button: Set Grid Parameters

The **Set Grid** command in the **Attributes Menu** of the main window and subwindows brings up a pop-up for setting grid parameters. This allows setting of the grid properties for the present display mode, for normal viewing or for printer output if the **Print Control Panel** is visible.

If the **Show** button is active, the grid will be visible. Otherwise, it will not be visible. 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. The **Store** and **Recall** buttons allow a set of grid parameters to be saved in an internal register, to be recalled as needed.

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.

There are text entry areas for the grid resolution and snap number. The resolution is the spacing of the grid lines, in microns. The snap number specifies the possible coordinates, i.e., the coordinate read by *Xic* when the pointer is in the vicinity. If the snap number is positive, it represents the number of snap locations per grid interval. For example, if 1, the snap locations are on the grid lines, if 2, the snap locations are on the grid lines and midway between the grid lines. If the snap number is negative, it represents the number of grid lines per snap line.

The line style editor allows the user to specify the patterning of the lines used to form the grid. There are three "radio buttons": **Solid**, **Dots**, and **Textured**. Choosing **Solid** will cause the grid to use continuous lines. The **Dots** option will use a grid consisting of a small dot at each grid point. If **Textured** is chosen, a user-specified patterned line will be used, and the editing windows become visible. 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.

Pressing the **Apply** button will actually save the new grid parameters in *Xic*, and redraw the windows.

The **Ctrl-G** key sequence is mapped by default to a command which also allows certain grid parameters to be set with a command-line interface. Thus, simple keyboard commands can be employed to change grid spacing and visibility. This may be a quicker than the pop-up for simple tasks such as turning the grid on and off. For example, to set the grid spacing to 2.5 microns, one types "**Ctrl-G** `2.5` **Enter**".

See also the description of the **!rg** and **!sg** commands. These can be used to save and restore the

grid from registers.

8.15 The Set Color Button: Set Colors Panel

The **Set Color** command brings up the **Color Selection** panel, which contains control sliders 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.

Near the top of the panel is a pull-down menu plus a "radio group" of three buttons. The three buttons allow the pull-down menu to contain different sets of entries. Each entry represents a color that can be adjusted.

The three buttons provide the following sets of menu entries:

Attributes button

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 button

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 button

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**.

In Windows, there are three sliders which control the red, green, and blue intensity, plus a sample window. The sample window is a drag source, so that colors can be transferred by dragging and dropping into the layer table. In most cases, the updated colors will not be visible until the screen is redrawn.

In Unix/Linux, there are six sliders which can adjust the red, green, and blue values, or the hue, saturation, and intensity values. Note that the sliders automatically track the current color setting. To the left is a color wheel, which provides another means of setting the hue and saturation, by dragging the small circular marker with the mouse. To the right of the color wheel is a rectangular area with a horizontal marker. Dragging the marker provides another means of changing the intensity.

Below the color wheel (Unix/Linux) is a rectangular area split into two fields. The left field shows the last color loaded into the color editor. The right color is the current color. One can drag the current color to one of the layer table layers, to set the color for that layer. This is cool, but the color of the current layer is automatically updated anyway. Note that except for pseudo-color visual modes ("8-bit" or 256 color displays) the main display will not show the new color until redrawn.

The color wheel, sample area, and saturation display are not shown in pseudo-color (256-color) modes, since in this case the colormap is not capable of rendering these. Rather, a large, unattractive gray area is presented instead.

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 names are obtained from the system `rgb.txt` file under Unix/Linux, or from an internal list in Windows. This file is assumed to exist in the following location:

FreeBSD/Linux	`/usr/X11R6/lib/X11/rgb.txt`
Solaris	`/usr/openwin/lib/X11/rgb.txt`

The changes to the underlying object or attribute color can be made by moving the sliders, selecting a color from the **Colors** menu, etc. The color change may or may not be immediately visible on screen. In some cases, one may have to force a redraw to see the color change. For example, if one of the **Prompt** colors is changed, one may have to move another window over the prompt line, then off, to send an expose event to the prompt line window which will force a redraw.

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.

8.16 The Set Fill Button: Fill Pattern Edit Panel

The **Set Fill** button brings up the **Fill Pattern Editor** panel, pre-loaded with the pattern of the current layer. This is employed to set or edit the fill pattern that is used with the layer. If the **Print Control** panel (induced by the **Print** button in the **File Menu**) is visible, the fill pattern set will be used for rendering the plot, if the plot driver supports definable fill patterns.

The left column of the display contains the **Pixel Editor** and **Sample** windows, which are loaded with the fill pattern of the current layer when the editor is started. The remaining windows contain default fill patterns. The first two pattern boxes always contain empty and solid fill, however the other 16 can contain any pattern. There are 64 internal pattern registers, 16 of which are displayed at any one time. A menu button below the pattern boxes allows each of the four banks of patterns to be displayed. Any of these pattern registers, with the exception of solid and empty, can be used for storage of other 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.

The following transfers are possible:

- The **Sample** box to/from any of the default pattern boxes except solid or empty, or any layer in the layer table. Drag patterns into the **Sample** area to allow editing with the **Pixel Editor**. Drag patterns into layers to set the pattern used for that layer. A pattern dragged into a default pattern box will replace the existing pattern with the one from the source.
- Any default pattern box to any layer in the layer table.
- Any layer in the layer table to any of the default pattern boxes except solid and empty.

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 do display patterns in the pop-up is the color of the current layer. Initiating a drag from a layer in the layer table 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. 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.

The **Outline** button, in the lower left of the pop-up, determines whether or not a solid outline is drawn around objects which use the pattern, once it is transferred to a layer. Set this to the desired state *before* dragging a pattern to a layer.

The pixel map can be one of four sizes: 8x8, 8x16, 16x8, or 16x16. The size selector button, located next to the **Outline** button, changes the size of the current pixel map. The size is automatically set when a pixel map is loaded into the editor.

The pixels in the pattern can be edited in the **Pixel Editor** box. The basic operation is to toggle the state of the 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 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.

8.17 The Edit Layers Button: Edit Layer Table

This button 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 to the right of all layers to put the new layer above the layers shown. 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.

Chapter 9

The Convert Menu: Data 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 **Edit** 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, except for the "Solaris" distribution, 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*.

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 `www.wrcad.com/oasis`.

WARNING: This is a prototype capability built from SEMI Draft Document 3626 2003/04/23 but adheres to the final published specification. The current implementation and documentation are preliminary.

That being said, the present status of OASIS support in Xic (and *XicII*) is fairly 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!

CIF

The CIF format, though a bit archaic, is still popular. *Xic* supports a number of selectable dialects and extensions.

The **Convert Menu** entry brings up a submenu 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 Control	`exprt`	**Conversion – Export**	Export files
Import Control	`imprt`	**Conversion – Import**	Import files
Convert	`convt`	**Conversion**	Direct conversions
Read Into Current	`readc`	none	Import data
Text Editor	`txted`	**Text Editor**	Text edit cell file
Edit Parameters	`adgds`	**Conversion Parameter Editor**	Edit GDSII layer map

The **Edit** command in the **Edit** 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 1.4 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.

9.1 Cell Name Aliasing

There is provision for modifying cell names as archive files are read, written, or format converted. The **Conversion – Import**, **Conversion — Export**, and **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 check box
: This check box is available in the **Conversion** − **Import** panel only. The state of the button tracks 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 **Edit** 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 **Conversion** and **Conversion** − **Import** panels, these text areas track the InCellNamePrefix and InCellNameSuffix variables. In the **Conversion** − **Export** panel, these text areas track the **OutCellNamePrefix** and **OutCellNameSuffix** variables.

  This will apply to files read with from 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 **Conversion** − **Import** and **Conversion** panels, these buttons track the state of the InToLower and InToUpper variables. In the **Conversion** − **Export** panel, these buttons track the state of the **OutToLower** and **OutToUpper** variables.

  These will apply to files read with the **Edit** command and equivalent, in addition to files opened from the panels and through script functions.

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 **Conversion** − **Import** and **Conversion** panels, the buttons track the InUseAlias variable, and in the **Conversion** − **Export** panel, the buttons track the OutUseAlias variable. Aliasing from the alias file is applied before any other name change.

  These will apply to files read with the **Edit** command and equivalent, in addition to files opened from the panels and through script functions.

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.

9.2 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.

9.3 Layer Names

The layer names used by *Xic* are CIF-style alphanumeric tokens of four characters or fewer. In addition, layers can have an optional long name, which has no restrictions on length or character type. Layers can be resolved by either name.

When working with GDSII and other files that use a numeric layer/datatype combination to designate layers, the default layer names are a special encoding described below. These layer/datatype combinations can be mapped into arbitrary *Xic* layers using the mapping constructs described in 9.5. If no such mapping is found, the default name will be used.

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, in the range 0–255. Similarly, the two right characters represent the datatype number. For example, layer 33, datatype 15 has the name "`210F`".

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 two hex digits of the appropriate field can be set to "`XX`". In the decimal representation, a single '`-`' replaces the appropriate digits. For example, "`0FXX`" and "`15,-`" equivalently specify layer number 15 and any datatype number.

9.4 Layer Filtering and Aliasing

The **Conversion - Import** and **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 table, when enabled by the **Use Layer Alias** button. The layer alias table 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 table will be applied to layers found in the file. This button tracks the state of the **UseLayerAlias** variable. The layer alias table is an internal table of layer mappings, which will be applied (if enabled) before the **Layer List** tests. The entries can not use wildcarding.

Edit Layer Aliases button

This button brings up a panel for editing the layer alias table. The layer alias table can also be modified with script functions, and the **!lal** command.

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 table 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. 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.

9.5 GDSII Layer Mapping

The GDSII file format does not use layer names. Instead, geometry can exist on a numbered layer and data type. Typically, the layer number and data type can be in the range 0–256, or 0–64 for some older versions of the GDSII specification. Here, the combination of a GDSII layer number and data type is referred to as a "specification".

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 GDSII parameter editor. 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 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 **GDSII: Map all unmapped datatypes...** button in the **Conversion – Import** panel produced by the **Import Control** button in the **Convert Menu**. When this variable is set (directly with the **!set** command, or by the button), the data type 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 data type = GDSII data type
        (use this layer)
```

```
    }
  }
}
```

If no suitable output mapping is found, *Xic* will look for special layer names. These names are four-character hexadecimal encoding of the layer number (first two characters) and data type (next two characters). Each field is 0–padded. In the situation where the data type is ignored (NoMapDatatypes is set), the data type characters are instead "XX". If a layer is found with a name matching (case-insensitive) the description just given for the unknown specification, and that layer has no input mapping, an input mapping will be created. The following pseudo-code illustrates the logic:

```
print layer_name  # NoMapDatatypes set: hhXX, otherwise hhhh
loop through existing Xic layers {
  if Xic layer name matches layer_name
    (use this layer)
}
```

If no existing layer is found that can be mapped to, a new layer will be created. The name of the new layer is as described above, which is:

NoMapDatatypes set
: The format is "*hh*XX" where *hh* is the hexadecimal value of the GDSII layer number, and XX is literal.

NoMapDatatypes not set
: The format is "*hhhh*" where the left two *h*'s represent the hexadecimal value of the GDSII layer number, and the right two *h*'s represent the hexadecimal value of the GDSII data type number.

The hex values are produced in upper case.

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.

9.6 The Export Control Button: Conversion – Export Panel

The **Conversion – Export** panel is brought up with the **Export Control** button in the **Convert Menu**. This panel allows various flags and variables that are specific to file output to be set, and also allows initiation of actual output operations. At the top of the panel are several buttons which set internal flags.

Strip For Export
: When the **Strip For Export** button is active, converted output will contain physical data only, and will contain no *Xic* extensions.

  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 the **Strip For Export** button is active, *Xic* will convert only the physical information when explicitly (i.e., using the operations from the **Conversion** – **Export** panel, and not the **Save** and **Save As** buttons) converting to an archive format. The **Strip For Export** setting also applies to the `ToCGX`, `ToCIF`, `ToGDS`, and `ToOASIS` script functions. This creates a file which should be an absolutely conventional physical layout file. The **Strip For Export** button should be active 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.

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 using the operations initiated from the **Conversion** – **Export** panel. This is the method by which certain layers can be eliminated from generated output. When this button is not active, all *Xic* layers will be converted.

This feature can also be enabled by setting the variable SkipInvisible with the **!set** command.

The remaining settings on this panel apply to all generated output, and not just to output initiated from this panel.

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.

Cell Name Mapping

This group of controls manages the cell name aliasing feature (see 9.1). This does not apply to native cell file output, and the prefix/suffix modifications apply only when the operation is initiated from this panel or a script function.

GDSII and OASIS: skip layers without Xic to GDS 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.

OASIS: 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.

NOTE: **compressed OASIS files are not compatible between the present release and release 2.5.57 and earlier**. Compressed OASIS files from the present release are compatible with other tools that follow the OASIS specification.

This mode can also be enabled by setting the boolean variable OasWriteCompressed with the **!set** command.

OASIS: 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.

OASIS: 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, which controls the use of REPETITION records in OASIS output. This variable can be set explicitly to fine-tune the use of REPETITION records. The check box sets/unsets this variable as a boolean.

OASIS: 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, 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.

9.6.1 CIF Dialect — Extension Styles

This section of the pop-up 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.

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 9.7).

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.

The default selections shown in the menus can be specified by setting the CIFoutStyle variable. When this variable is set, changing the selections in the menus will update the value of this variable. If this variable is *not* set, then reading in a CIF file will alter the menu selections so as to write a file in the same dialect. Otherwise, the defaults (the first choice in each menu) are used.

9.6.2 Output Format

The **Output Format** menu provides a selection of file formats to be used for written output. The choices are Xic (native), and the supported archive formats.

The **Conversion Scale Factor** provides an entry area where a scale factor to be applied during the operation can be entered. Values of 0.001 through 1000.0 are acceptable. This will apply to output initiated from this panel and script functions only.

The operation is actually initiated with the **Write File** button. The name of the output file will be prompted for on the prompt line. The **Dismiss** button retires the panel.

Cell files can also be written to disk using the **Save** and **Save As** commands in the **File Menu**. However, if scaling or the **Strip For Export** or **Skip Invisible Layers** features are needed, the file must be generated from this panel.

The following choices are available for converted output format.

Xic

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* format.

Before the operation starts, 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. Pressing **Esc** will abort the operation.

After the operation is complete, the user is prompted for the name of a library file which will be written in the current directory. This file will have references to each of the new files created, with the top-level symbol 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. Pressing **Esc** will skip creating the library file.

When generating an archive file and an error occurs. the archive file will normally be deleted. However, if the variable KeepBadArchive is set (with the **!set** command) the output file will be given a "`.BAD`" extension and retained. This file should be considered corrupt, but may be useful for diagnostics.

CGX

With this choice, the current editing cell and its descendents are written to a CGX 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.

CIF

With this choice, the current editing cell and its descendents are written to a CIF 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 style entry areas described above.

GDSII

This choice creates a GDSII (Stream) file of the current editing cell and its descendents. 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 data types used are as given in the technology file.

Xic will ensure that symbol 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 Parameters** command. By default, a layer needed for output that does not have a mapping will terminate the operation. However, if the **GDSII: skip layers without map** button described above is set, or equivalently the variable NoGdsMapOk is set (with the **!set** command), then such layers will be ignored (producing no output).

OASIS

This choice creates an OASIS file of the current editing cell and its descendents. The name of the file for the OASIS output is requested from the user. The layer numbers and data types 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 Parameters** command. By default, a layer needed for output that does not have a mapping will terminate the operation. However, if the **GDSII and OASIS: skip layers without map** button described above is set, or equivalently the variable NoGdsMapOk is set (with the **!set** command), then such layers will be ignored (producing no output).

9.7 The Import Control Button: Conversion – Import Panel

The **Conversion – Import** panel is brought up by the **Import Control** button in the **Convert Menu**. The panel controls certain features used when reading cell data files. Reading of files into *Xic* memory can also be initiated through this panel.

The panel contains a number of check boxes and option menus, as follows. Unless otherwise noted, these settings apply to all input, for example through use of the **Edit** command, and not just input initiated from this panel.

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 the 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 (below).

This state can also be enabled by setting the boolean variable NoAskOverwrite with the **!set** command.

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**.

Merge boxes and coincident objects while reading

When this button is on, boxes are merged together, if possible, as files are being read in. Overlapping boxes are clipped and/or merged. This applies to box objects only, and not polygons (even rectangular ones) or wires.

In addition, objects (not subcells) that are coincident with an identical object will be deleted. This applies to all geometrical objects and not just boxes.

This mode can also be enabled by setting the boolean variable MergeInput with the **!set** command.

The merging of boxes is suppressed if the NoMergeBoxes variable is set (with the **!set** command). Coincident boxes will still be deleted, however.

Skip testing for badly formed polygons
When set, the reentrancy test for polygons is skipped while an input file is being read. 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.

Skip testing for coincident objects
When set, testing for coincident objects (but not subcells) is skipped while reading input. Coincident subcells are always reported in the log file. This has no effect on the merging of coincident objects if the **Merge boxes** button is active.

This mode can also be enabled by setting the boolean variable NoCoincCheck with the **!set** command.

Skip testing for empty symbols
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 symbols in the hierarchy is suppressed. An "empty symbol" is a symbol that contains no physical or electrical objects or subcells. It is possible to check for empty symbols at any time with the **!empties** command.

This mode can also be enabled by setting the boolean variable NoCheckEmpties with the **!set** command.

Cell Name Mapping
This group of controls manages the cell name aliasing feature (see 9.1). 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 symbols. The prefix/suffix modifications are applied only in input initiated from this panel or script functions.

GDSII: map all unmapped datatypes to same Xic layer
This variable affects only the creation of new layers when a GDSII 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 mode can also be enabled by setting the boolean variable NoMapDatatypes with the **!set** command.

CIF: how to resolve 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.

The layer change module allows layer filtering and/or mapping to be applied during the read operation (see 9.4). These actions can be performed when:

- Input is being read from an archive file.
- The operation was initiated from the **Conversion** − **Import** or **Conversion** panel, or from on of the `FromArchive`, `Edit`, or `OpenFile` script functions.
- The physical-mode part of the file is being read. The **Layer list** applies only to physical data.

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 and script functions only.

The **Read File** button will prompt the user for a file to read into *Xic*, in the manner of the **Edit** command. However for archive files, scaling and/or layer filtering may be applied to the cells read from the file. This, and the script commands `Edit` and `OpenCell`, use the scale factor provided in the text area or with the `SetConvertScale` script function, and the **Layer list** (if enabled, archive files only). Other methods of opening a cell do not provide scaling or layer filtering.

The **Dismiss** button removes the pop-up from the screen.

9.8 The Convert Button: Conversion Panel

The **Conversion** panel is brought up by the **Convert** button in the **Convert Menu**. This panel is a front end to a number of direct conversion functions which translate an input file into output of another format. These are direct conversions, i.e., the data are converted directly and do not enter the database. This means that there are no memory limitations, and as 64-bit file offsets are used, 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 **Conversion** − **Export** panel.

The input and output file types are selected from the two drop-down menus. The content of the output format menu may change as the input format is selected, as not all permutations are supported. Conversion between any of the archive formats is possible, as is conversion from an archive format to native symbol files. Binary archive files can be translated to a human-readable text representation by selecting "**text**" in the output format menu, which may be useful for debugging corrupted files. The text representation of GDSII files can be back-converted to GDSII format.

There are several check boxes which control details of the output format. The four OASIS controls also appear in the **Conversion − Export** panel.

OASIS: 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.

NOTE: **compressed OASIS files are not compatible between the present release and release 2.5.57 and earlier**. Compressed OASIS files from the present release are compatible with other tools that follow the OASIS specification.

This mode can also be enabled by setting the boolean variable OasWriteCompressed with the **!set** command.

OASIS: 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.

OASIS: 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, which controls the use of REPETITION records in OASIS output. This variable can be set explicitly to fine-tune the use of REPETITION records. The check box sets/unsets this variable as a boolean.

OASIS: 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, 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.

The remaining two check boxes affect the ascii text output from OASIS files.

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.

The **Cell Name Mapping** group of controls manages the cell name aliasing feature (see 9.1).

The layer change module allows layer filtering and/or mapping to be applied during the read operation (see 9.4). These actions can be performed when:

- Input is being read from an archive file.
- The operation was initiated from the **Conversion – Import** or **Conversion** panel, or from on of the `FromArchive`, `Edit`, or `OpenFile` script functions.
- The physical-mode part of the file is being read. The **Layer list** applies only to physical data.

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.

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 file. 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.

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.

If the **No Empties** button is active, empty subcells and their references will not appear in output. Use of layer filtering and windowing with clipping and not flattening may generate empty cells, and use of this button prevents these from appearing in output.

When using windowing or flattening, best performance is obtained when the input file is *not* compressed. Resolving offsets into the file requires a lot of decompression which can substantially increase conversion time.

The translation is initiated with the **Convert** button. The user will be prompted for the name of the input file, and then the name of the output file, or directory for native files.

If the input and output formats are both GDSII or both CGX, and there is no windowing, flattening, scaling, or layer filtering, no parsing will occur. Instead, the file is directly copied, possibly with compression or decompression. The "`.gz`" suffix on the input or output file names indicates whether compression/decompression is applied. Unlike some versions of the standard `gzip`/`gunzip` utility, this is applicable to files larger than 2Gb in *Xic* releases that support 64-bit file offsets.

When generating an archive file and an error occurs. the archive file will normally be deleted. However, if the variable `KeepBadArchive` is set (with the **!set** command) the output file will be given a "`.BAD`" extension and retained. This file should be considered corrupt, but may be useful for diagnostics.

9.8.1 Conversion: Notes and Limitations

When scaling, only the physical (not electrical) data are scaled.

If the input file contains multiple top-level cells, all are translated, using the specified parameters.

When **Use Window** or **Flatten Hierarchy** is set, only physical data are converted, i.e., there will

be no electrical data in the resulting file. The format of the new file will conform to the settings in the **Conversion – Export** panel.

When **Use Window** or **Flatten Hierarchy** is set, an internal symbol table is used to keep track of the hierarchy (similar to skeleton mode). There is a consequent chance of running into memory limitations with the hugest files. Labels are ignored in these modes.

When clipping, wires that require clipping are converted to polygons.

If the input file is in CIF format, and symbol 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.

When converting CIF files to GDSII or OASIS, if the CIF layer is found in the layer table and it has a GDSII output (layer and datatype number) mapping, that mapping will be used. Otherwise, if the CIF layer name is a four-digit hex number, it will be interpreted "LLDD" to obtain the GDSII layer and datatype numbers. Otherwise, a new layer number and datatype will be internally generated. New layer numbers start at 1 and increment, with datatype number 255.

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*.

The GDSII directives absolute magnification, absolute angle, and absolute path width are not supported in *Xic* 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.

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 the NumToTxtCells variable can be used to limit the number of cells written. 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 *Xic* 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-0.2` 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.

When the selected output format is **Xic**, the input will be converted to a number of native symbol 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 **Edit** command, and then using the **Conversion – Export** panel to generate the *Xic* files.

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 (by up to twenty percent).

9.9 The Read Into Current Button: Read In Data from Another Cell

The purpose of this command is to copy the electrical or physical part of another cell into the present cell. If in physical mode, only the physical part is copied, and similar for electrical mode. This command is useful for merging data if a physical layout and electrical schematic exist in separate cells.

The target for the copy is always the cell shown in the main window. After pressing **Read Into Current**, the user is prompted for the name of a cell or file to use as input. If the name entered matches a cell in memory, the copy is performed and the command exits. Otherwise, the name is expected to be a file. If the file exists, the internal symbol table context is switched and the file is opened with an empty temporary symbol table. The file can be an *Xic* cell, or an archive file. If the top level cell in the hierarchy matches the name of the current cell or the file contains only one cell, the copy is performed, otherwise the user is prompted for the name of a cell in the hierarchy to copy. The user can bring up the **Cells Listing** panel (from the **File Menu**) to see a listing. After using cut/paste or typing the name of a cell at the prompt, the copy is performed, the temporary symbol table and its contents are destroyed, and the command exits.

There is no undo capability for this operation.

9.10 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 1.4.

9.11 The Edit Parameters Button: Edit Conversion Parameters

This command brings up a text editing window displaying the LongName, Description, StreamIn, StreamOut, and NoDrcDatatype lines for the current layer, as they would appear in the technology file. Changing the current layer will update the listing to the parameters for the new current layer. The user can add new lines or modify existing lines as desired. The **Save Tech** button in the **Attributes Menu** can be used to write a new technology file containing the modifications.

To add a specification, select the desired keyword in the **Keywords** menu of the editor. The user will then be asked to enter the associated text on the prompt line.

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.

The syntax for the keywords is described below. The LongName and Description fields of layers are multi-purpose and do not exclusively pertain to conversion functions. Their inclusion in this editor is due to lack of a better place for this functionality. The remaining parameters are specific to conversion, and in fact GDSII and OASIS conversion only.

LongName *longname*
: This will set (or reset) the long name 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 argument.

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.

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 255. 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 "0-255". 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.

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 $0 - 255$. 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, the objects will be added to the GDSII or OASIS file on each of the GDSII layers specified. There is no default.

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.

9.12 Skeleton Mode

In this mode, which applies to archive files, no geometry is saved while the file is opened, only the cell hierarchy is saved in memory. However, when geometry is needed, it will be read directly from the archive file. This feature is to accommodate huge files which would require more than the available memory or produce excessive page swapping when read in the usual way. The feature is probably most useful in server mode, or in scripts where only a small part of the design needs to be in memory at one time. It is suitable for analysis operations, but typically not for synthesis, as changes made are inherently volatile.

Warning: do not use gzipped (`.gz`) files in skeleton mode. Although these will "work", performance will be poor due to the need to decompress for each file access.

Skeleton mode is established by setting the boolean variable SkeletonMode with the **!set** command or equivalent. While this variable is set, archive files will be read in the manner described, other files will be read normally. Any number of skeletal hierarchies can exist in memory simultaneously (earlier versions allowed only one).

The only purpose of the SkeletonMode variable is to provide a means to read files skeletally using the normal input functions. The variable is ignored in operations once the hierarchy is in memory.

A second way by which skeletal hierarchies can be read into memory is through use of an archive context and the `CxSetupSkeleton` function. This can be more conservative of memory.

When the current cell is skeletal, functions such as `SkelReadGeom` can be applied. The skeletal status flag is kept in the cell, and can be displayed with the **Info** command.

In skeleton mode, only physical data are read. Unlike normal files, the electrical data structure is not created, saving memory space. Thus, none of the functions that require electrical data will work with skeletal cells.

To use skeleton mode, it is necessary to set a clipping window. The clipping window ensures that only the geometry that overlaps the clipping window with nonzero intersection area, when reflected into the coordinates of the top-level cell, will be read into memory. The clipping window is set with the following script function:

`SkelSetDataWin`(*l*, *b*, *r*, *t*)

The arguments are the left, bottom, right, top coordinates, in microns, relative to the origin of the current cell. If scaling has been applied when reading the skeleton, the arguments must be the scaled coordinates. This and all script functions can be executed directly from the prompt area by preceding the function call with "`!!`" (two exclamation points).

Geometry is actually read into memory with the following script function:

`SkelReadGeom`(*depth*)

The *depth* specifies the depth into the hierarchy to read. This can be -1 to specify reading all of the hierarchy. Once this function has been called, all of the standard physical-mode operations can be performed on the cell data, in the regular way.

The geometry presently in memory can be cleared with the script function

`SkelClearGeom`()

Since the database is cleared before reading in new data with `SkelReadGeom`, this function is not generally needed.

Although in general geometry is not kept persistently in memory, relaxing this can improve performance in some cases. This feature is enabled by the following variable:

SkelSaveNum (integer >= 0)

If a cell contains this many geometrical objects or fewer, it is retained in memory. The assumption is that there are small cells that are instantiated in many places, and it is more efficient to cache these cells rather than having to update from the file each time the content is needed. This variable should be set to some small number. If set to 0, this feature is effectively disabled.

Finally, it is possible to write the windowed data directly to a new output file with the following function:

`SkelWriteGeom`(*depth*, *filename*, *flat*, *clip*)

This function will write the cells and objects that have nonzero area overlap with the current skeleton window to a file. The type of file written is implied by the filename extension. The remaining arguments are flags that indicate hierarchy flattening and object clipping. See the function description for complete information.

The following sequence should appear in scripts to change the skeleton mode data window.

```
SkelSetDataWin()
SkelReadGeom()
```

Virtually all of the time these two functions will be used together in the sequence shown. Following this sequence, all operations are available, though the hierarchy contains only the objects which reflect into the data window in the coordinate system of the top-level cell. Although objects can be added/edited in the normal manner, the changes will be lost when the file data are re-read. A disk file can be saved from skeleton mode, but be careful to not clobber the original file with a file with partial content. The original file will be saved with a "`.bak`" extension *once*.

These script functions have no effect on normal cells.

Once the hierarchy has been read and a clipping window set, the user can use any appropriate script functions or physical-only operations including DRC. Only the geometry whose bounding box overlaps the clipping window with nonzero area will "exist". The process of stepping through a huge design piece by piece in this way is expected to be much faster than the straightforward method, which may not even be possible due to memory limitation.

This page intentionally left blank.

Chapter 10

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**
Clear Errors	`clear`	none	Erase error indicators
Set Defaults	`limit`	none	Set error limits
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
Check In Foreground	`check`	none	Test rules in foreground
Check In Background	`bgchk`	none	Test rules in background
Check In Region	`point`	none	Test rules in region
Dump Error File	`erdmp`	none	Dump errors to file
Show Errors	`next`	subwindow	Sequentially display errors
Query Errors	`query`	none	Print error messages
Edit Rules	`dimen`	**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.

10.1 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 layer 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.

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 following operators are accepted in layer expressions:

& or *	intersection
| or +	union
!	inversion
^	exclusive-or
−	and-not, i.e., $A - B = A\&!B$
`and`	synonym for &
`or`	synonym for |
`not`	synonym for !

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.

`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.

`edges`(real *incr*, layer_exp *layer*, int *mode*)
: This creates an edge list, similar to the `EdgesZ` script function.

`manhattanize`(real *dimen*, layer_exp *layer*)
: 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. This is similar to the **!manh** command, 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.

`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 where there is no overlap between 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.

Examples:

```
!layer M2 = M2 & box(100, 100, 200, 200)
```

This clips M2 to the given box.

```
!layer M2 = bloat(5, M2)
```

This bloats the M2 geometry by 5 microns.

10.2 Built-In Design Rules

Design rules are specified in layer blocks of the technology file by a keyword, followed by an optional source region specification, followed by parameters. There can be an arbitrary string following the parameters. This string is not used internally, but is printed with rule violation messages and usually contains a brief description and reference number for the rule.

The rules make use of layer expressions. A layer expression can be a single layer name, or a more complicated expression. 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
```

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.

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.

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.

10.2.1 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.

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 errors 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*] [*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 flagged as an error in the top level cell.

10.2.2 Area Rules

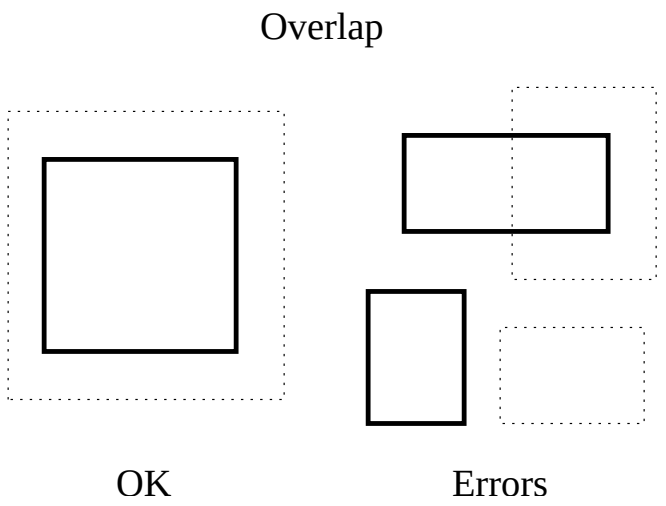
The following are the per-area tests, and are applied to the area of each source figure, for each object in the pseudo-flat representation.

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

Figure 10.1: The Overlap test. The present figure (solid) must be completely covered by figures resulting from evaluating the expression argument (dotted).



true at every point of every object on the present layer. This is illustrated in Figure 10.1, 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 10.2 illustrates use of this keyword, for no Region and an expression consisting of a single layer.

NoOverlap **Rule**

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 10.3, for no Region and an expression consisting of a single layer.

AnyOverlap **Rule**

Syntax: `AnyOverlap` [`Region` *region_expr*] *expression* [*string*]

Figure 10.2: The IfOverlap test. The present figure (solid) can not be partially covered by figures resulting from evaluating the expression argument (dotted).

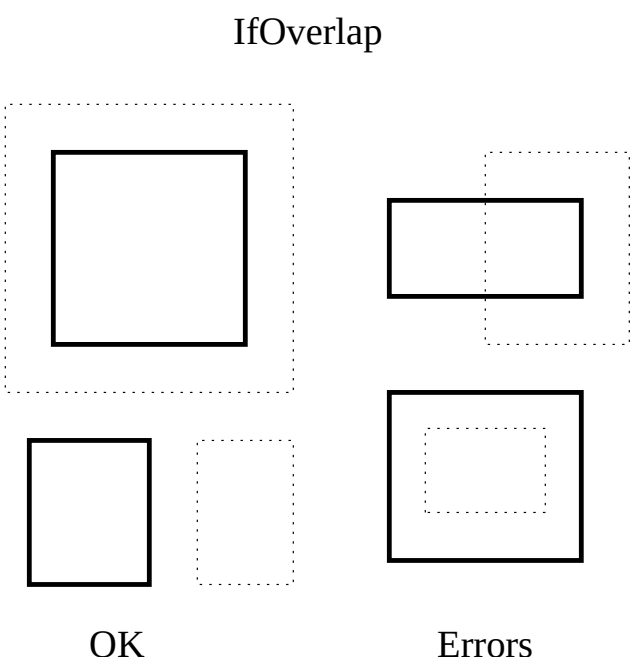


Figure 10.3: The NoOverlap test. The present figure (solid) can not intersect with figures resulting from evaluating the expression argument (dotted).

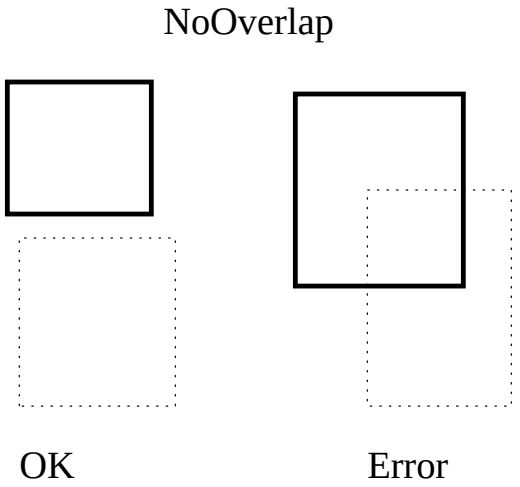
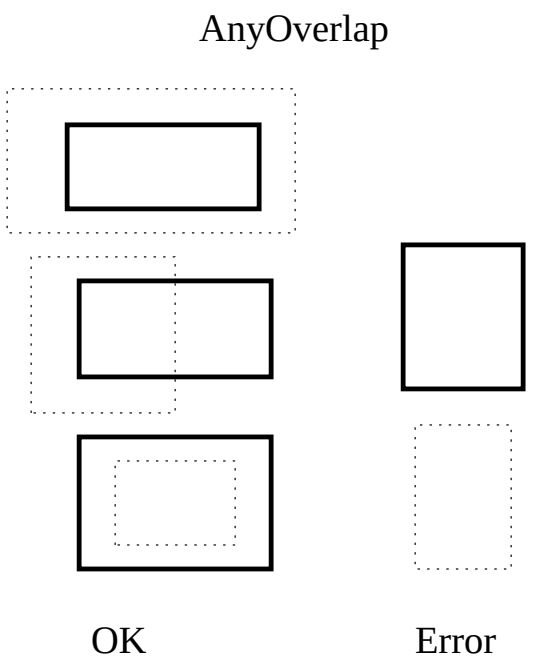


Figure 10.4: The AnyOverlap test. The present figure (solid) must be partially or fully covered by figures resulting from evaluating the expression argument (dotted).



The AnyOverlap test signals an error if any source figure has no intersection area with the figures associated with the *expression*. This is illustrated in Figure 10.4, for no Region and an expression consisting of a single layer.

PartOverlap **Rule**

Syntax: `PartOverlap` [`Region` *region_expr*] *expression* [*string*]

The PartOverlap test signals an error if any source figure is either completely covered or completely uncovered by the figures associated with the *expression*. This is illustrated in Figure 10.5, for no Region and an expression consisting of a single layer.

AnyNoOverlap **Rule**

Syntax: `AnyNoOverlap` [`Region` *region_expr*] *expression* [*string*]

The AnyNoOverlap test signals an error if any source figure is completely covered by the figures associated with the *expression*. This is illustrated in Figure 10.6, for no Region and an expression consisting of a single layer.

The returns from the various Overlap tests are summarized in the table below.

Figure 10.5: The PartOverlap test. The present figure (solid) must be partially covered by figures resulting from evaluating the expression argument (dotted).

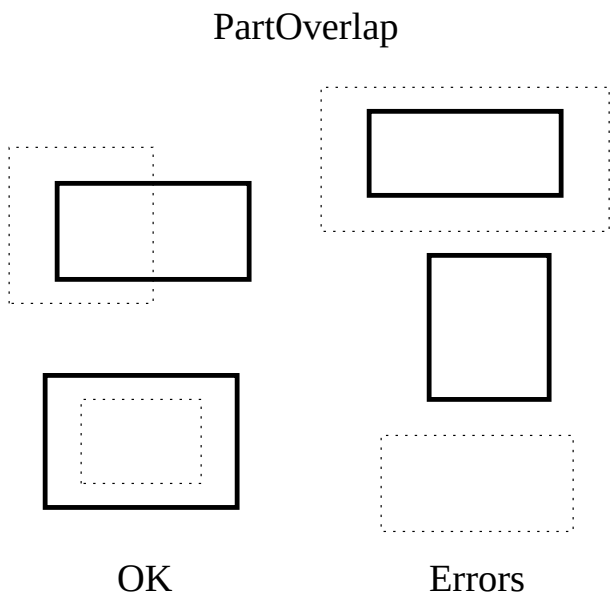
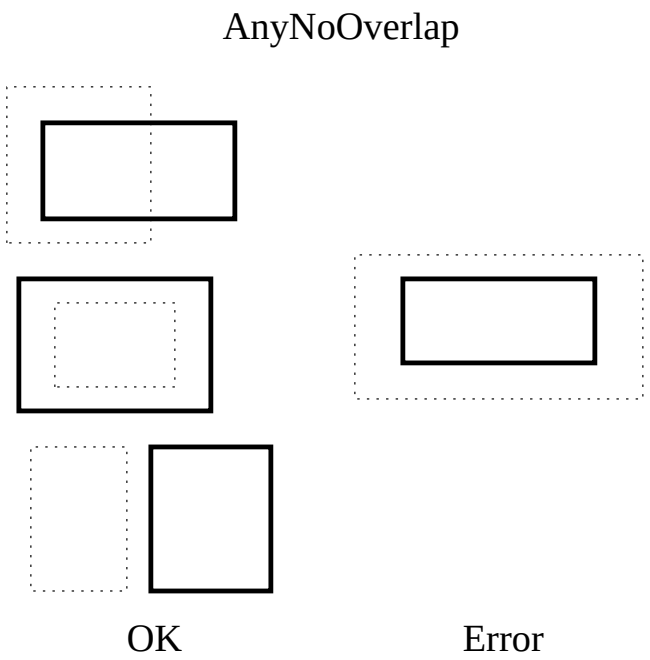


Figure 10.6: The AnyNoOverlap test. The present figure (solid) must be partially uncovered by figures resulting from evaluating the expression argument (dotted).



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*]

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 less than the test value a DRC error 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.

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 error 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.

10.2.3 Edge Rules

The remaining 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 of each source figure.

In the descriptions, the "target" is the set of figures associated with the *expression* supplied to the rule, for those rules that take an *expression*.

MinEdgeLength **Rule**

Syntax: `MinEdgeLength` [`Region` *region_expr*] *expression length* [*string*]

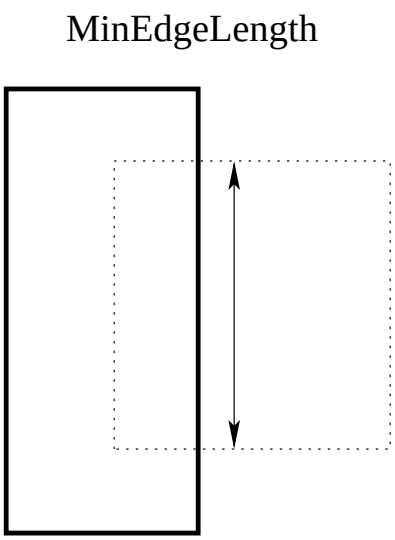
This test checks the length of the edges where source and target figures intersect. For each edge of the source figure, the parts of the edge where *expression* is true on both sides of the edge are considered. If the length of the part is less than the given *length*, an error is flagged.

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:

Figure 10.7: 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.




```
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
```

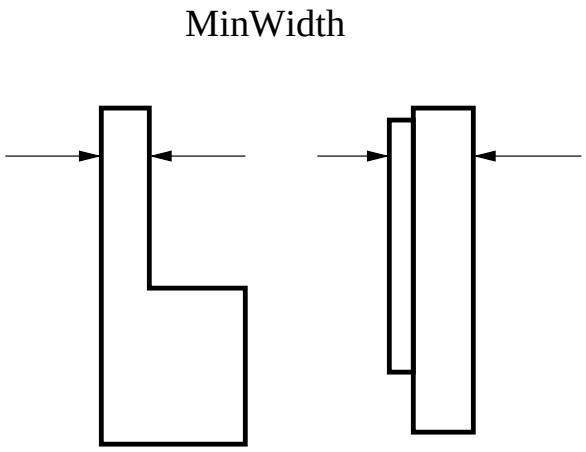
MinWidth **Rule**

Syntax: `MinWidth` [`Region` *region_expr*] *width_in_microns* [*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 10.8 illustrates the test performed under this keyword, for no Region.

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.

Figure 10.8: 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` [`Region` *region_expr*] *space_in_microns* [*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 10.9 illustrates the test performed under this keyword, for no Region.

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.

MinSpaceTo **Rule**

Syntax: `MinSpaceTo` [`Region` *region_expr*] *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 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 error, but touching figures will generate an error. Figure 10.10 illustrates the test performed under this keyword, for no Region and an *expression* consisting of a single layer.

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 10.9: The MinSpace test. The edge-to-edge spacing between regions on the present layer must not be less than the given dimension.

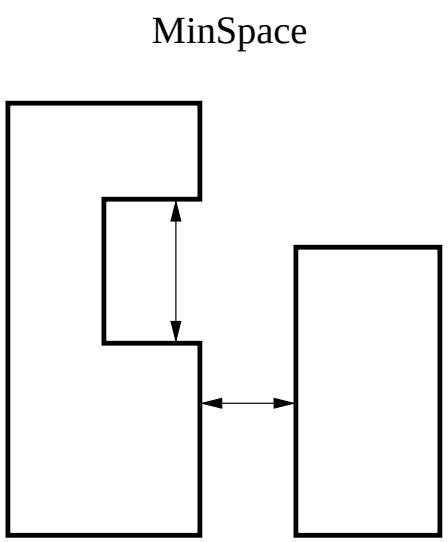


Figure 10.10: The MinSpaceTo test. The minimum edge-to-edge spacing between regions of the present layer (solid) and the argument layer (dotted) must not be less than the given dimension.

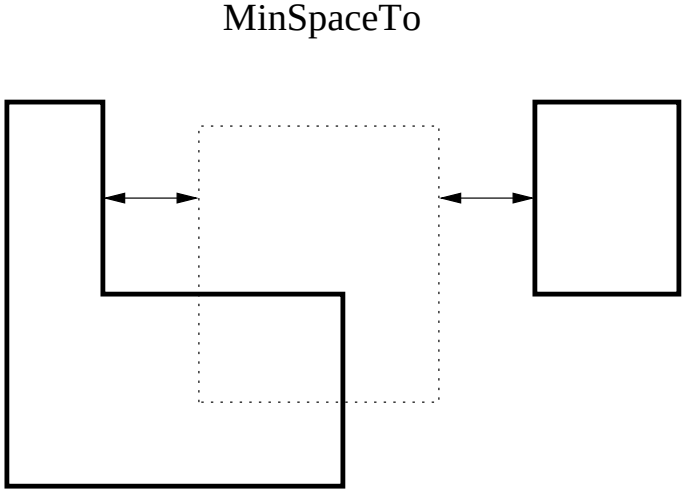
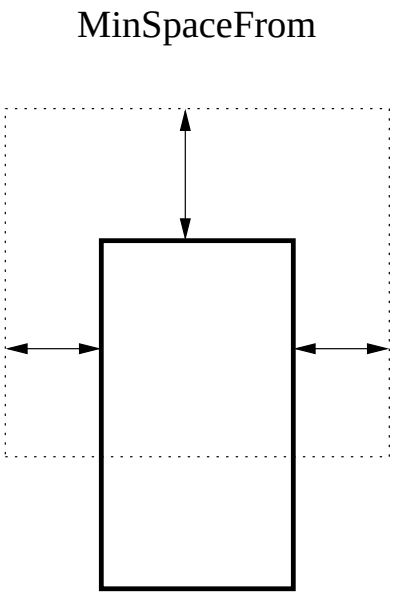


Figure 10.11: The MinSpaceFrom test. The rule is violated if the projection from the current layer, if any, is less than the supplied dimension.



MinSpaceFrom **Rule**

Syntax: `MinSpaceFrom` [`Region` *region_expr*] *expression dimension_in_microns* [*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 10.11 illustrates the test performed under this keyword, for no Region and an *expression* consisting of a single layer.

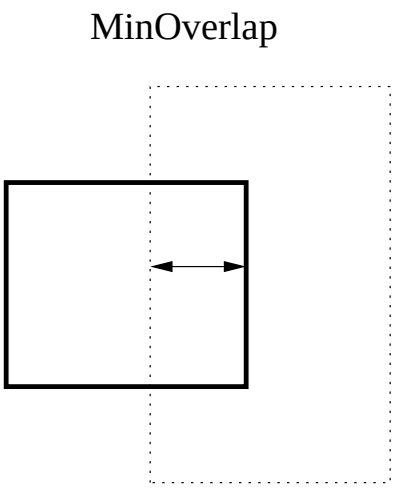
This is in many cases redundant with the MinNoOverlap test (see below) if applied to the result of the *expression*, if the *expression* is simply a layer name.

MinOverlap **Rule**

Syntax: `MinOverlap` [`Region` *region_expr*] *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 10.12 illustrates the test performed under this keyword, for no Region and an *expression* consisting of a single layer.

Figure 10.12: 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.



MinNoOverlap **Rule**

Syntax: `MinNoOverlap` [`Region` *region_expr*] *expression dimension_in_microns* [*string*]

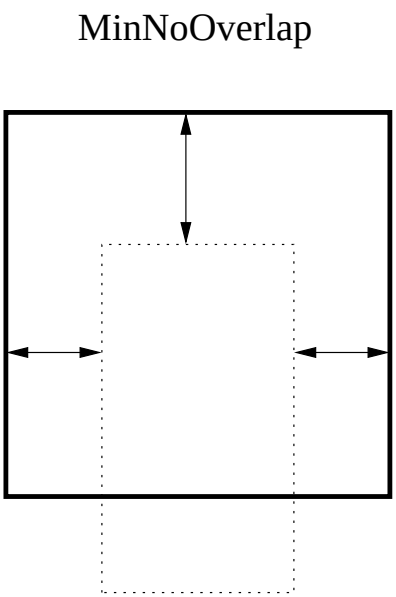
For the parts of each edge of the source that are not coincident or overlapping with source-compatible figures or with target figures which extend into the interior of the source figure, a rectangle extending normally from the edge into the source figure a distance given by the dimension is constructed. The test fails if the constructed rectangle has nonzero intersection area with target figures. Figure 10.13 illustrates the test performed under this keyword, for no Region and an *expression* consisting of a single layer.

10.3 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`

Figure 10.13: 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.



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 EdgeMin and EdgeMax 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 EdgeMin dimension, and less than or equal to the EdgeMax 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 EdgeMin and EdgeMax. 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
EdgeMin 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
```

10.4 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 10.3.

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.

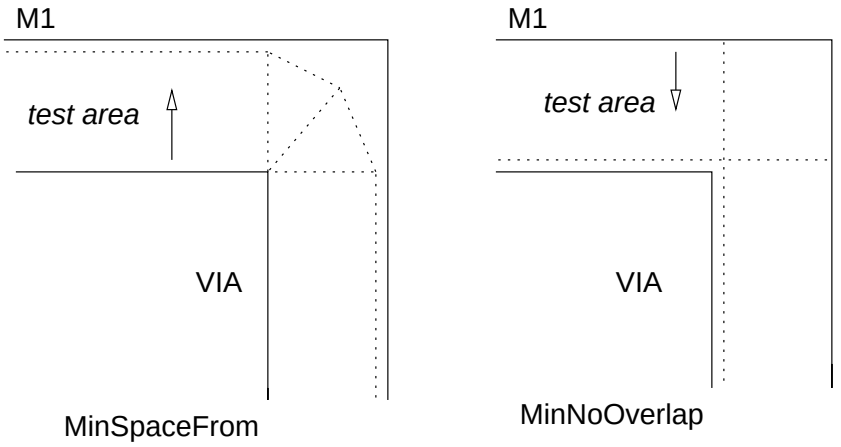
In reciprocal rules, such as MinSpaceTo, which specifies the minimum distance between objects on two different layers, it is often questioned whether the rule should be specified in each layer. The answer is no, although no real harm is done if it is specified in both layers, though both specifications 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 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 10.14.

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.

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.

Figure 10.14: The MinSpaceFrom and MinNoOverlap tests differ in the treatment of the corner regions projecting outward from the central feature, as shown.



10.5 The Clear Errors Button: Clear Error List

Pressing the **Clear Errors** button 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.

10.6 The Set Defaults Button: Set Default Limits

The **Set Defaults** button brings up a panel which allows the user to set limits and other parameters used in design rule checking. The first limit is on the number of errors reported in batch mode checking (with the **Check In Foreground** or **Check In Background** buttons). If this limit is reached, the checking aborts. 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 is active). 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. Of course, interactive testing can be switched off entirely with the **Set Interactive** button. The number 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 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 error recording level. The default is to record only one error per object. With many errors, this can cut evaluation time, and may be useful for a first pass. The second choice outputs a maximum of one error of each type (i.e., corresponding to each DRC keyword) per object. The third choice will output all errors found. This can lead to redundancy, as certain violations may be reported for each *edge* of the offending object.

10.7 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 Parameters** command in the **Convert 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.

10.8 The Enable Interactive Button: Set Interactive Checking

When this button 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 ignore the error.

10.9 The No Pop Up Errors Button: Suppress Error Report

When this toggle button is set, errors produced in interactive DRC do not cause error 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.

One can add "DrcNoPopup y" to the attributes section of the technology file to suppress the error pop-up automatically.

10.10 The Check In Foreground Button: Check Region or Cell

The **Check In Foreground** button initiates design rule checking on a region or cell. The user is requested to point to the corners of a rectangle defining the region to be checked. Simply clicking will cause checking of the objects under the pointer. Holding and dragging will cause checking of all objects which overlap the ghost-drawn rectangle, when the button is released. If the user presses and holds without moving the pointer for a brief period, the ghost-drawn rectangle corner will be attached to the pointer, and pressing a second time will complete the operation. Instead of a mouse button press, pressing the **Enter** key will cause the entire area of the current cell to be checked. All objects in the current cell and its subcells which overlap the specified region are checked. Errors are recorded in a file named "drcerror.log" which is written in the current directory. Additionally, violating objects are marked, and the error highlighted. During checking, button and key presses are locked out, however pressing **Ctrl-C** will abort the checking.

10.11 The Check In Background Button: Initiate Check in Background

This command initiates design rule checking on a region or cell. Operation is similar to the **Check In Foreground** button (below), however a new background process is created to actually perform the checking. This process will create an errors file in the current directory named `drcerror.log.`*PID* where *PID* is the process id of the spawned process. A pop-up message will appear in *Xic* when a spawned process completes. 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.

The user is requested to point to the corners of a rectangle defining the region to be checked. Simply clicking will cause checking of the objects under the pointer. Holding and dragging will cause checking of all objects which overlap the ghost-drawn rectangle, when the button is released. If the user presses and holds without moving the pointer for a brief period, the ghost-drawn rectangle corner will be attached to the pointer, and pressing a second time will complete the operation. Instead of a mouse button press, pressing **Enter** will cause the entire area of the current cell to be checked. All objects in the current cell and its subcells which overlap the specified region are checked. Unlike the **Check In Foreground** command, errors are not marked on-screen. The spawned process can be stopped or killed using the job control functionality of the user's shell. There can be multiple spawned processes executing concurrently.

Under Windows, this works by executing a batch-mode *Xic* process in the background. However, the entire cell is always checked, any rectangle specification is ignored.

10.12 The Check In Region Button: Check Objects

When the **Check In Region** button 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.

10.13 The Dump Error File Button: Save Errors to File

This command allows the user to dump a file containing the error messages for the currently visible errors. The user is first given the chance to change the file name. 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 is written to a temporary file which is erased later, which is convenient if the user only wants to view the text.

10.14 The Show Errors Button: Show Next Error

After batch rule checking (using the **Check In Foreground** or **Check In Background** commands) is performed, errors may be viewed sequentially with this command. The **Show Errors** button sets a mode where pressing the **PageDown** key will display the first and subsequent errors in a subwindow. 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**. 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. The log file must report errors for the current cell. If a file is successfully loaded, the **Show Errors** button will be shown active.

The functionality is maintained until the **Show Errors** button is selected a second time, making it inactive. 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. Note that in the subwindow, only the current error is highlighted, whereas in other windows, all errors are highlighted, if the errors were generated in the current session, otherwise no errors are shown.

Show Errors mode is terminated if a new cell is opened for editing, including **Push** and **Pop**, and upon switching to electrical mode. However, If the command is restarted later with the same cell and error file, the first error shown will be the error visible when the command was terminated.

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.

10.15 The Query Errors Button: Print Error Text

When the **Query Errors** button 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.

10.16 The Edit Rules Button: Rule Editor Panel

The **Edit Rules** button 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 prompting for new values for the parameters for the selected rule, after which the rule is deselected. Editing mode can be exited with the **Esc** key. 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 prompt the user for parameters required for that rule, then add the new rule to the list of rules for the current layer, or will replace an existing MinWidth or MinSpace rule if that type is added. 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 submenu 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.

Chapter 11

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 also 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**
Select Groups	`gnsel`	none	Select groups
Quick Paths	`qpath`	none	Highlight conducting paths
Show Paths	`paths`	none	Highlight conducting paths
Show Groups	`group`	none	Show group numbers
Show Nodes	`nodes`	none	Show node numbers
Show Terminals	`tshow`	none	Show terminals
Edit Terminals	`tedit`	none	Edit terminal names/locations
Find Terminals	`tfind`	subwindow	Locate terminal
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`	none	Save physical/electrical comparison
Compare Devices	`cmpar`	none	Compare/set devices
Compute Params	`cmput`	none	Compute electrical parameters
Extract RLC	`exrlc`	**RLC Extraction**	Extract electrical parameters
Edit Extraction	`adelp`	**Parameter Editor**	Edit extraction parameters

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. The technology file specifies the information necessary to perform these operations.

Grouping assigns a group number to each collection of connected conducting objects. It identifies connections through vias and contact layers, and takes into account the operations specified in "Conductor Exclude" directives. Extraction is the identification of physical devices and subcircuits, and establishment of the connections. This is the most compute-intensive task, since connections can be made through and between subcells, the entire cell hierarchy must be processed. Extraction also provides measurement of parameter values based on geometry. Association provides the linkage to the elements of the schematic. During association, a file named `associate.log` is produced, which contains messages emitted during the association process.

If the extraction cannot complete due to an error, or if an important error is identified such as problems in the technology file setup, a window containing the `associate.log` file will appear. This will contain messages which should help identify the problem. Note that lack of 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.

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 can be placed (with the **Edit Terminals** command), however this is not usually necessary. Manually placing these terminals may assist in the association process, and may be needed in some cases to resolve ambiguity.

The association algorithm works by comparing electrical and physical devices, subcircuits, and nodes, finding matches by evaluating a context score. The pair with the highest unambiguous context score are taken as duals. If the matching is ambiguous, a permutation search is performed, and the permutation with the lowest number of non-resolutions is kept.

Caveat: It may not be possible to associate some topologies, in which case there is no means at present to coerce convergence. Example files that don't associate would be welcomed for debugging purposes.

Many of the devices in the device library have physical terminal extensions included in their node property definitions. 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 vcc terminals. These have no explicit physical implementation, but are an implicit part of the wiring net as they assign connection indices to the net to which they are connected.

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 **Show Terminals** button. 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 moved to their proper locations in the physical layout.

Xic has 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). This algorithm, coupled with knowledge of the contact locations provided by device and subcircuit extraction, provides the basis for the association.

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

- similar Conductor or ground plane layers touching.

- Contact and Conductor or ground plane layers touching.
- 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.

Any subcell whose electrical part has no connections is checked for connectivity, and used to reduce the group numbering in the parent cell. Thus, the conductor group extends through via cells, for example. Vias and similar wiring cells should have no electrical terminals, and should not be placed in the schematic.

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. In particular, for an inductor, three new layers are used. One layer defines the area of the inductor device. A second layer is used to identify the contacts, and a third layer is used in a "Conductor Exclude" directive to separate the inductor conductor into two groups.

Separate commands are available for generating netlist files from the physical and electrical data. The **Dump LVS** command performs the layout vs. schematic comparison, and prints errors in a file and on-screen.

In extraction there is a capability for automatically flattening certain physical subcells in the extracted netlist. The cells to be automatically flattened must have names that begin with a common prefix, such as "$$". The flattening is enabled by setting the variable FlattenPrefix to the prefix, e.g.,

```
!set FlattenPrefix $$
```

When a flattenable subcell is found during extraction, all devices and sub-subcells in the subcell are linked into the parent cell for extraction and LVS purposes. References to these cells will disappear from the **Dump Phys Netlist** and **Dump LVS** listings, unless the boolean variable PnetListAll is set, in which case they are listed.

In many EDA systems, extensive use is made of special cells for MOS devices, for example. These may be automatically generated or pulled from some library. In use, it is more than cumbersome to identify each such device with a subcell in the schematic. The facility above allows the user to avoid this.

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.

The core of the extraction 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). This algorithm, coupled with knowledge of the terminal locations, provides the basis for the extraction. This "grouping" is performed when needed, and along with related extraction functions accounts for the delay and activity noted in response to many of the command buttons and other operations. Once a cell is processed, it is not regrouped unless the cell is modified. The **Show Groups** button in the **Extract Menu** performs grouping if necessary, and displays the group numbers.

Each group (conductor net) is assigned a number. While the **Show Groups** button is active, these numbers are printed on-screen near the conducting objects.

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 the Extraction Setup topic.

11.1 Extraction Setup and Configuration

Use of the extraction features requires setting certain keywords and data blocks in the technology file. This section describes these entries. There are two types of entries: keyword descriptions in the various physical layer blocks, and Device blocks, which appear after the layer blocks in the technology file. 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 11.1.3 provides more information on this capability.

11.1.1 Layer Block Keywords

This section describes the keyword entries which appear in layer blocks which relate to extraction. These define the conductor layers which are involved in grouping, identify vias between conductors, etc. Further, geometrical electrical data can be assigned to layers, which will be used for parameter extraction. All of these settings can be entered with the **Edit Extraction** command in the **Extract 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 10.1. 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
: 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.

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`.

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.

Symbolic
If this keyword is present in a layer block, the layer will *not* be included in the cross section displays produced with the **Cross Section** command (in the **View Menu**).

The following keywords can appear only in physical layer fields, and they specify the computation of electrical parameters associated with the layer. These parameters can be displayed while boxes are created (with the **boxes** command), and are used in the **Compute Params** command. The results from

these models should be considered as an approximation. No account is taken of the "real" geometry, just the simple assumption of a non-featured dielectric and ground plane underneath the conductor, for the capacitor and transmission line models.

Resis[tance] *ohms_per_square*
: The single parameter is a floating point number giving the ohms per square value of the material. This is used in extracting the resistance of resistor devices.

Cap[acitance] *units_per_sq_micron* [*units_per_micron*]
: This enables computation of the capacitance of a conductor group. 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.

Tran[line] *line_thick line_pene gp_thick gp_pene diel_thick diel_const*
: This keyword will invoke a microstrip model which computes the transmission line parameters along the long dimension of a newly created box. The parameters are all floating point numbers, and dimensions are in microns, except for the dielectric constant which is dimensionless. The penetration depths are included (the model is actually for a superconducting microstrip line) and can be set to some small value for normal metals. The algorithm is the same as used in the public domain sline program available as source code from the Whiteley Research web site. These parameters are used for inductance extraction. The extraction assumes that the layer is over a continuous ground plane, separated by a constant-thickness dielectric.

The following keywords are used by the FastCap/Fasthenry extraction interface. Each keyword takes a single real value argument.

Thickness *thickness*
: This keyword supplies the film thickness of the corresponding deposited film. The *thickness* is given in microns.

At most one of the following two keywords (Rho and Sigma) should be used. These apply to layers with the Conductor attribute given explicitly or implicitly.

Rho *resistivity*
: This keyword supplies the resistivity, in MKS units, of the corresponding conducting film.

Sigma *conductivity*
: This keyword supplies the conductivity, in MKS units, of the corresponding conducting film. This is converted to resistivity (1.0/*conductivity*) internally.

EpsRel *diel_constant*
: This keyword supplies the relative dielectric constant of insulating layers. An insulating layer must also have the Via keyword given, or must be a passivation layer. A passivation layer is a single layer that must be above all Via layers which has the EpsRel given and a nonzero Thickness given.

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.

11.1.2 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.

Name *device_name*
: The *device_name* names the device, which should match a name in the device library (`device.lib`) file. Two or more Device blocks can use the same name. This would be necessary, for example, if there are two resistor layers in a process. A Device block would be needed to describe resistors on each layer. This keyword is mandatory.

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 prefix should be unique among all devices. The first letter of the prefix should match the expectations of the SPICE simulator to be used.

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).

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 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.

  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
: The minimum height or width of the trapezoids which describe the Body region. This is intended 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. Then, 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 11.1.

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 the layer defining the device body.

Inductance
Extract the inductance value (see below).

Mutual_Inductance
Extract the mutual inductance value. This is not yet implemented.

LVS *measure_name* [*spice_name*]
This keyword instructs *Xic* to perform a parameter comparison as part of LVS. The *measure_name* is one of the names used for a Measure statement in the Device block. 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.

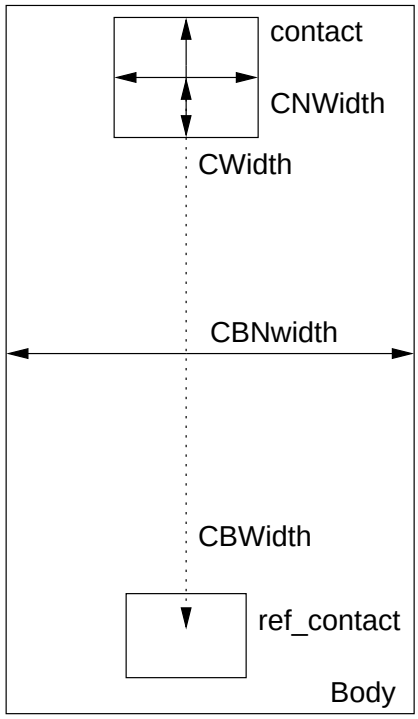
Examples:

```
Measure Area BodyArea*1e-12
LVS Area area

Measure Resistance Resistance
LVS Resistance
```

Any number of LVS lines can appear in a Device block.

Figure 11.1: The distances returned by the various width measurement keywords to the Device block Measure line.



Spice *specification_args*

The Spice keyword specifies the format for the SPICE output part of the listing from the **Dump 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 Model, Value, and Param lines are used internally in the **Compare Devices** command when setting the properties of an electrical device from the physical device. The Spice line is used in **Dump Phys Netlist** output and internally by the **!exset** and **Source Physical** commands.

Cmput *specification_args*
: This specifies the format used in printing the device parameter in the **Compute Params** command. 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 setting the properties of the electrical device from parameters extracted from the physical equivalent, with the **!exset**, **Source Physical**, or **Compare Devices** commands. The format is the same as is described for the Spice line, however the escapes %model%, %value%, and %param% are not recognized.

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.

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.

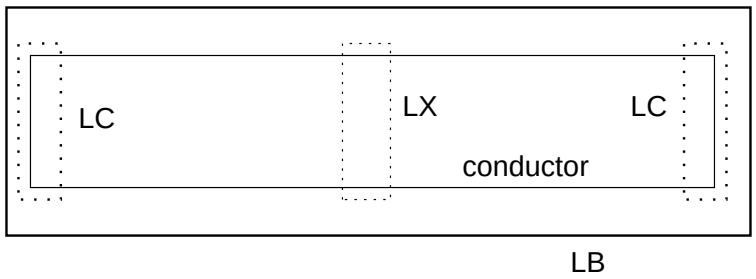
RLSolverGridPoints
: **Value:** integer 10–100000.
When not tiling (RLSolverTryTile is not set), this sets the number of grid points used for resistance/inductance extraction. This number will be the same for all device structures, so that computation time per device is nearly constant. Higher numbers 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.

The resistor solver is accessed through the Device block Measure keyword "Resistance", for example:

```
Device
Name res
...
Measure value Resistance
End
```

Figure 11.2: 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.



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 a Resistance specification, and if found, that value is taken as the ohms per square. If not found, 1 ohm per square is assumed.

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 top conductor layer. The Tranline parameters must be given for the conductor layer in the technology file.

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.

11.1.3 Format Library File

Xic provides a mechanism for user-specified formatting of physical and electrical netlist output and LVS results. 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.

This information is PRELIMINARY. Status:
The LVS formatting is not implemented yet.

The electrical netlist has basic implementation.
The physical netlist has basic implementation.

The `xic_format_lib` file should be placed in the library search path, typically in the startup directory. An example file is provided with the distribution, which can be used as a starting point for customization.

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.

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

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.

11.2 Extraction Methodology and Overview

To use the extraction capability, one typically first designs the circuit in electrical mode, then produces a corresponding layout in physical mode. The **Show Groups** command is a useful diagnostic aid, as it shows the group numbers of conducting objects (a group number is analogous to a circuit node number). Objects which have been partitioned, as have been clipped due to a Conductor Exclude directive, have multiple group assignments, and the partition boundaries are shown in the **Show Groups** command.

When the **Select Groups** button is active, clicking on an object or a partition will select the entire group to which the object or partition belongs, if the command was entered while in physical mode.

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 vcc 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, or equivalently with the **!source** command. 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, or equivalently the **!exset** command will 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 **Compare Devices** button can update the properties of the electrical device from parameters measured for the corresponding physical device, on a per-device basis. The **Compute Params** button will print the result of a similar parameter measurement, but will not set properties.

The **Import** button in 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 performs the layout vs. schematic comparison, and prints errors in a file and 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.

These commands are described in the following sections.

11.3 The Select Groups Button: Select Groups/Nodes

When the **Select Groups** button is active, clicking on a conducting object will highlight the conductor group in physical views and the corresponding electrical connections in electrical views. This command works in either mode, i.e., either a schematic location or a conducting object can be clicked on in the editing windows.

When active, typing a number followed by **Enter** will select the group corresponding to the number entered.

11.4 The Quick Paths Button: Show Electrical Connectivity

This command is functionally similar to the **Show Paths** command, but operates outside of the extraction system. For complex designs, the initial extraction setup can be quite time consuming. Using this command will avoid this delay. Of course, if extraction is needed for other purposes, the **Show Paths** should can be used, as it is a bit quicker (once extraction is set up) and has a more powerful path determination algorithm.

The **Quick Paths** 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** command. For example, consider MOS devices. Since, the source and drain are connected to a common area of the "active" layer, which is a Conductor, the simple algorithm used in this command will interpret the source and drain as being connected (since it does not recognize the MOS device). 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 **Extraction Parameter Editor**.

If layer-specific selection is enabled, one must click on an object on the current layer to select the path. Otherwise, clicking on any conducting object will highlight the connected objects. Only one path is shown.

The boolean variable LayerSearchUp, if set, will enable searching for layers from bottom to top if set, rather than the default top to bottom. This affects the conductor chosen if the user clicks over more than one conductor layer, and layer-specific mode is not active.

If a dark-field ground plane has been specified in the technology file, the implied connectivity to ground is similar to that in force for the extraction system, though can be specified with the QpathGroundPlane variable. This can be set with the **!set** command to an integer in the range 0–2.

0 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 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 Paths** command will never use the inverted ground plane.

11.5 The Show Paths Button: Show Conducting Paths

The **Show Paths** button allows the selection and display of conducting paths, through different levels of the hierarchy. The user can select the expansion depth of objects to be included. Clicking on an object above this depth will cause all electrically connected objects above the depth to be shown highlighted. The clicked-on object must be on a layer with the Conductor keyword specified.

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.

When a path is displayed, the user can click on two objects in the path, and only the "subpath" 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 subpath. When a subpath is displayed, clicking on any non-selected object will display the full path containing that object.

While the command is active, the expansion depth can be changed with the '−', '+', '**n**', '**a**', and digit keys, similar to the **Expand** command in the **View Menu**. The present path will be redrawn with the new depth, if possible (the original object must be above the new depth). The path can be made to blink by pressing the '**h**' key. The **Show Paths** button is typically used to visually check the connectivity of wire nets in a layout.

There is a functionally identical **Quick Paths** command which works outside of the extraction system, thus avoiding the sometimes long setup time. However, this variation s limitations, as discussed in that topic.

If layer-specific selection is enabled, one must click on an object on the current layer to select the path. Otherwise, clicking on any conducting object will highlight the connected objects. Only one path is shown.

The boolean variable LayerSearchUp, if set, will enable searching for layers from bottom to top, rather than the default top to bottom. This affects the conductor chosen if the user clicks over more than one conductor layer, and layer-specific mode is not active. 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).

11.6 The Show Groups Button: Show Conductor Groups

The **Show Groups** command causes the group number of each conducting object to be displayed in physical windows. This is similar to the **Show Nodes** command, and is mutually exclusive with that command.

11.7 The Show Nodes Button: Show Node Numbers

The **Show Nodes** command causes the associated node name of each conducting object to be displayed in physical windows. This is similar to the **Show Groups** command, and is mutually exclusive with that command.

11.8 The Show Terminals Button: Show All Terminals

This command 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.

11.9 The Edit Terminals Button: Place Connection Terminal

This command makes visible the current cell's contact terminals, if any have been assigned with the **subct** command in the electrical side menu. These terminals are automatically placed during association (if possible) however this command allows manual placement and editing of the properties of the terminals.

To select a terminal for editing, the user clicks on or drags over the terminal to be edited. The selected terminal will blink, and a pop-up will appear. The pop-up allows the terminal name to be changed, and the layer binding to be set. If not explicitly named, the names default to "_*N*", *N* being an integer. The terminal (node) names are stored in the node property strings. These names can also be set when the terminal is created, while in electrical mode, with the **subct** command. In instances of the cell, the terminal name becomes a concatenation of the instance name and the parent cell terminal name, such as `X0_1` (defaults) or `Xbuffer_output` (applied values). The layer binding is a hint to the association function as to which layer the terminal should be associated with. This is important if the terminal is placed over objects on several different layers. In some cases, the association function will override the hint and assign the terminal to an object on another layer, if the association is "sure" that this is correct.

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. Clicking on a terminal with the **Shift** key held will attach it to the pointer, and clicking a second time will place the terminal at the new location. The terminal can also be moved by dragging. 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). There may be topologies where manual placement of terminals enables the correct association.

The cell formal terminals have a 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 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 properties editor, 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 Show Terminals command is a benign way to force a recalculation of all associations.

11.10 The Find Terminals Button: Locate Terminal

This command is for finding the locations of terminals, and works in physical and electrical mode. When the **Find Terminals** button is pressed, a pop-up appears which solicits a terminal name. After the name is entered, a subwindow appears with the view centered on the given terminal. Either physical or electrical mode instantiations of the terminal can be seen by switching between modes in the subwindow. The location of the terminal is at the exact center of the window, so one can zoom in or out with the numeric keypad '+' and '–' keys. The terminal is actually visible only if terminals are being displayed, with the **terms** command in electrical mode, or the **Show Terminals** command in physical mode.

11.11 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.

clear
: If set, the electrical part of a cell is cleared before reading the SPICE input. This implies **create**.

This functionality is also available through the **!source** command available from the prompt line.

If **create** is set, 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** command or **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 vcc 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, 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, the device model must be available. The semiconductor devices can be resolved (as p or n type) if the referenced model is not in the input file if 1) the model is in the model library database, or 2) the model name starts with "n" or "p".

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

11.12 The Source Physical Button: Update Electrical From Physical

This command 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.

This functionality is also available through the **!exset** command available from the prompt line.

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

11.13 The Dump Phys Netlist Button: Dump Physical Netlist

This command 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.

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 only when **net** and **spice** are enabled, respectively.

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 1000 units correspond to one micron.

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.

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.

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, 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.

list all cells	PnetListAll
list bottom-up	PnetBottomUp
net	NoPnet
show geometry	PnetShowGeometry
devs	NoPnetDevs
spice	NoPnetSpice
include wire cap	PnetIncludeWireCap

11.14 The Dump Elec Netlist Button: Dump Electrical Netlist

This button 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.

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.

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, 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.

list bottom-up	EnetBottomUp
net	NoEnet
spice	EnetSpice

If the variable CheckSolitary is set with the **!set** command then warnings are issued if nodes are encountered with one connection only.

11.15 The Dump LVS Button: Test Layout vs. Schematic

This function compares the netlists obtained from the physical and electrical data for 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.

11.15.1 LVS Output File Format

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 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 artrue:

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.

11.16 The Compare Devices Button: Compare Extracted Parameters

When this command 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 the **Shift** key is held while clicking on a device, the values for the electrical device are updated from the extracted physical parameters. The command is available in both electrical and physical modes. The **!find** command can be used to highlight extractable devices in the physical layout.

The physical device must be specified in a Device block, and have at least one parameter with the LVS keyword specified.

11.17 The Compute Params Button: Electrical Parameter Extraction

The **Compute Params** command performs device parameter extraction. It has two basic modes.

If the user clicks on the body area of a device defined with a Device block in the technology file, an extraction is performed and the results are printed. The format of the output is defined in the Device

block following the Cmput keyword. The **!find** command can be used to highlight extractable devices in the physical layout.

If the user drags, or presses and holds, the behavior of the command is different. A ghost rectangle is shown, and the per-square parameter from the current layer adjusted for the rectangle aspect ratio is printed when the user releases button 1 or clicks a second time (thus completing the rectangle). The electrical specifications are given in the technology file, and may be edited with the **Edit Extraction** command. The **Compute Params** command is especially useful when creating devices such as resistors. It is also useful for extracting electrical parameters from existing layouts.

These results can also be shown when the user creates a box with the **boxes** (side menu) command on a layer with electrical parameters specified.

11.18 The Extract RLC Button: Extract Parameters

This command brings up a panel which controls the FastCap/FastHenry interface.

FastCap and FastHenry are three-dimensional extraction programs for capacitance and resistance/inductance, respectively. Both programs are available in the Free Software Archive on the Whiteley Research web site:

```
http://www.srware.com/freestuff.html/
```

These versions must be used with this interface, though other versions of FastHenry may be used if none of the conductors is superconducting.

11.18.1 The FastCap/FastHenry Interface

Xic contains an interface to modified versions of the FastCap capacitance and FastHenry resistance/inductance extraction programs. These programs are available on the Whiteley Research web site. The original versions of FastCap and FastHenry were developed at MIT, and are publicly available in source code form on the Internet.

`FastCap`
: The modified version (`fastcap-2.0wr`) allows all input to be supplied as a single file. The MIT version requires input from multiple input files. The modified version is required to process output from *Xic* conveniently.

`FastHenry`
: The modified version (`fasthenry-3.0wr`) handles superconducting conductors, otherwise it is functionally identical to the MIT version.

Distributions of both programs provide detailed user manuals.

The interface allows extraction from planar structures likely to be encountered in integrated circuits. It is not a general purpose interface, though it can perhaps be used as a starting point when addressing general problems. It is best suited to extracting from a small collection of conducting objects. The user should be aware that both FastCap and FastHenry can become monsters with regard to memory use and execution time.

The interface is controlled from the **RLC Extraction** panel which is brought up by the **Extract RLC** button in the **Extract Menu**. In addition, there is the **!fx** prompt line command which allows most of the interface to be controlled through text input. There are a number of variables (see C.16) which are used by the interface. Most of these are maintained through entry areas in the **RLC Extraction** panel, but can also be set with the **!set** command.

Briefly, selected conducting objects are added to an internal database associated with the interface. The physical properties of the layers are given in the technology file. The interface uses this information to create a three-dimensional representation of the selected objects and surrounding insulation layers. This representation can be "refined" through a partition editor, to reduce granularity in the simulation, thus improving accuracy. Locations of assumed connecting terminals, needed for FastHenry, can be defined. Output can be dumped in the form of FastCap or FastHenry input files, or as a "`.sav`" file to save the geometry for a future run. FastCap and FastHenry can be run directly, and the results, after some minimal post-processing, will be displayed (and saved in a file).

FastCap/FastHenry Interface Limitations

The Fastcap/Fasthenry interface, though it accepts non-Manhattan geometry, may not be useful for some data sets. A problem arises when there are thin, angled conductor strips. The Manhattan partitioning required to adequately represent the geometry may produce so many FastHenry segments or FastCap panels that the simulation time is very long. Future work will involve developing non-Manhattan partitioning techniques to address this issue.

The FastHenry output uses one filament per segment. Future work will be directed toward making use of the filamentation capability provided in the FastHenry program. At present, `.Default` statements can be added to FastHenry output by hand as a partial remedy. This may be particularly useful with superconductors, in the z-direction to better model penetration depth.

Technology File Setup

To use the interface, the technology file must be set up appropriately. This involves adding keywords to the physical layer blocks of the technology file, which define physical attributes of the materials. This can be done by editing the technology file with a text editor, or through use of the **Extraction Parameter Editor** from within *Xic*. In the latter approach, the **Save Tech** command can be used to dump an updated technology file.

The interface will determine a sequence of layers from the technology file. The first layer is a conductor, and is assumed to contact the substrate. Additional layers alternate between insulators and conductors.

It may be necessary to create "dummy" layers in order to accommodate layer sequences that are not as above. For example, suppose that one has a substrate covered by dielectric, followed by a conductor. One can define a dummy conductor layer between the substrate and the insulator, but not create any geometry on this layer. The effective sequence is then that desired, and the preprocessor will correctly handle input. Note, however, that the effective thickness of the dielectric is the specified thickness plus the thickness of the dummy conductor (which is arbitrary but must be larger than zero). This is due to the assumption of planarization, where "holes" in the dummy layer are filled with the covering dielectric.

Only layers that are visible will be used in the interface. The visibility of layers can be toggled from the layer menu by clicking with mouse button 2. The default layer visibility is set with the Invisible technology file layer block keyword. Making a layer invisible is a quick way to eliminate it from the data set used in the interface. When a conductor layer is made invisible, the via layers associated with the

conductor will also be eliminated from consideration.

Layers that are recognized as conductors must have the following properties:

1. The layer must be visible.
2. The layer must have the conductor attribute. This will be true if any of the Conductor, Routing, Contact, or one of the ground plane keywords has been applied.
3. The layer does *not* have the Contact keyword applied. These layers are not used in the layer sequence, but geometry from Contact layers will be merged with the geometry from the target layer of the Contact specification, if that layer is in the sequence.
4. The layer must have the Thickness keyword applied, with a positive layer thickness value.

These keywords a described fully in 11.1.1.

Layers that are recognized as insulating via layers must have the following properties:

1. The layer must be visible.
2. The layer must have the Via keyword applied, and the target conductors must be in the list of recognized conductors.
3. The layer must have the Thickness keyword applied, with a positive layer thickness value.

There is also a possible passivation insulator layer, that will cover the top conductor. If it exists, this layer is assumed to be everywhere present, and any actual patterning is ignored. A passivation layer must have the following properties:

1. The layer must be visible.
2. The layer must not have a conductor attribute or the Via keyword applied.
3. The layer must appear in the layer menu to the right of (above) the topmost conductor used in the sequence.
4. The layer must have the Thickness keyword applied, with a positive layer thickness value.
5. The Layer must have the EpsRel keyword given, with a dielectric constant value larger than 1.0.

The ordering of the layers in the technology file (and the layer menu) is generally independent of the ordering of the layers when sequenced by the interface. There are two exceptions:

1. The passivation layer, if one appears, must be defined after all of the conductor layers in the technology file (or appear above the conductor layers in the layer menu).
2. The lowest conductor must be listed before the topmost conductor in the technology file (or appear below the topmost conductor in the layer menu).

The ordering is otherwise determined from the Via references, which is not necessarily the order in which the layers appear in the technology file or layer menu.

The interface will perform the layer sequencing when a FastCap or FastHenry file is created, or whenever the sequencing is needed. A table of layer ordering is printed in a FastCap/FastHenry input file created.

Below is a listing of the physical layer block keywords that are used by the interface.

Conductor
Routing
Contact
GroundPlane
GroundPlaneClear
: Each of these keywords applies a conductor attribute to the layer. A layer must have the conductor attribute to be recognized as a conductor by the interface.

Via
: This keyword indicates that the layer is an insulator, and is used to make contact between two conductors.

Thickness
: This specifies the physical thickness of the layer, in microns. Only layers with nonzero thickness will be used by the interface.

Rho/Sigma
: Either of these parameters can be applied to a conductor layer to provide a resistivity/conductivity value, in MKS units. This value will be used in the FastHenry input. If not specified, FastHenry will use its own default, which is the conductivity of copper. Only one of these should be applied per layer.

Lambda
: This provides the superconducting London penetration depth of a conductor, and if nonzero will signal FastHenry to treat the conductor as a superconductor. The value must be given in microns.

EpsRel
: This provides a relative dielectric constant to insulating layers.

DarkField
: This is implied by Via and GroundPlaneClear, and indicates that the physical material is the inverse of the patterning shown on-screen.

These keywords can be applied to the layer blocks in the technology file with a text editor, or can be introduced with the **Extraction Parameter Editor** in the **Extract Menu**.

The conductor layers can be given a resistivity or conductivity with the Rho and Sigma keywords, respectively. These are used by FastHenry (FastCap assumes perfect conductors). 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 constant of the insulators is specified with the EpsRel parameter. If a Via layer is not given an EpsRel specification, 3.8 is assumed. A passivation layer must have EpsRel specified. These values are used only by FastCap.

The relative dielectric constant of the substrate can be altered by setting the SubstrateEps attribute keyword in the technology file. The keyword can be placed in the attributes section of the technology file, after all layer blocks and Device blocks, and the relative dielectric constant should follow the keyword. If not set, the value for silicon (11.9) is assumed. This parameter can be set with the **Extraction Parameter Editor**, or by editing the technology file. This is used only by FastCap.

Geometry Specification

It is very important to understand how the interface interprets the layer sequence. The interface assumes that the technology is planarized, thus for a conducting layer, places where the conductor is absent are assumed to be filled with the dielectric material which is *above* the conductor. A side effect is that if a conductor is entirely absent, the effective dielectric thickness is actually the dielectric layer thickness plus the thickness of the missing conductor.

It is also worth noting that DarkField layers are polarity-inverted in the interface. If a DarkField layer is missing from the display, within the interface the layer is assumed to be everywhere present. Via layers are always DarkField, as are GroundPlaneClear conductors.

Objects are added to the interface database with the **Save Selections** button in the **RLC Extraction** panel, or with the corresponding **!fx save** command. The interface database is entirely separate from the geometry database displayed on-screen. Once a figure is placed in the interface database, any modification or deletion of the object shown on-screen will have no effect on the saved representation in the interface. There is no way to modify the shape of the object within the interface once it has been saved. The interface database can be cleared with the **Clear Saved** button in the panel.

When an object is saved in the interface, its geometry is split into a trapezoid representation, which is "Manhattanized". Thus, non-Manhattan geometry is supported, but through this approximation. The granularity of the Manhattanization is controlled by the FxMinRectSize variable or the corresponding text entry field in the **RLC Extraction** panel. The Manhattanized trapezoid list is saved in one long list of rectangles from all objects saved for the layer. When the geometry is reconstructed prior to processing for FastCap/FastHenry, the entire list will be joined into a minimal set of disjoint Manhattan polygons, each of which is a separate conducting object as seen by FastCap or FastHenry. Thus, overlapping and touching figures in the main database are effectively merged in the interface database.

Geometry from layers with the Contact technology file keyword is added to the layer that is the target of the Contact specification, if it is included in the layer set.

Only conducting objects are actually added to the interface database. Attempts to add other objects will (silently) have no effect. The related insulators are automatically extracted from the layout when the conductor rectangle lists are joined into polygons. First, a list of vias is extracted, Manhattanized, and clipped to the associated conductor polygons. The vias are extracted to all depths in the cell hierarchy. The resulting list of rectangular vias represent the connection points between conductor layers.

The vias are then represented internally by a column of conductor material of the same type as the upper conductor, which extends down through the insulator and makes contact to the lower conductor. Thus. planarization is maintained at the via.

For FastCap, the dielectric interface planes are then defined. These are the planes at the bottom of each conductor layer, outside of the conducting objects, which represent the interface between the lower dielectric material and the upper dielectric material (recall that the upper dielectric material fills the parts of the conductor layer where the conductor is absent). These plane areas extend outside of the bounding box of all conductors by some distance, which is given (in microns) with the FxPlaneBloat variable, or equivalently with the text field in the **RLC Extraction** panel. This distance should be chosen so that the truncation of the dielectric is far enough away from the conductors that the error introduced in capacitance calculation is minimal, which generally means that the distance is large compared to layer thicknesses.

The bounding box is the minimum-sized rectangle that encloses all conducting figures added to the interface. For conducting layers that are DarkField, the layers are polarity-inverted within this rectangle extended by the FxPlaneBloat setting. Thus, a dark field ground plane, for example, will extend beyond

the boundaries of the other conductors by this amount. When using DarkField conductors, one must bear in mind that they are always present, as they will be invisible in the display.

Saving and Recalling Context

Once a set of objects has been saved, partitions refined and FastHenry terminals added (these operations are described below), pressing the **Clear Saved** or exiting *Xic* will send all of the hard work to bit-heaven, unless the data set has been saved. This is accomplished with the **Dump Saved** button in the **RLC Extraction** panel.

Every data set has a name. The data set can be given a name in the text field provided in the **RLC Extraction** panel, or if no name is supplied, the name will be "`unnamed`". This name will be used as the file base name when files are created from the panel for the current data set.

In the case of the **Dump Saved** operation, the default file name is the data set name with a "`.sav`" extension. This is a text file, in a unique format, that contains all of the layer, geometry, partitioning, and other information currently saved in the interface. The file is not expected to be edited or used otherwise by the user, so at this point the format will not be documented, though it should not be too difficult to reverse engineer.

The data set can be recalled with the **Recall Saved** button in the **RLC Extraction** panel. When this is engaged, any data set presently in memory will be overwritten (so save it first if necessary), and the saved data set will be read into the interface. In addition, the editing context will switch to a dummy cell that contains the objects saved in the interface. Unlike in the normal operation, the objects as displayed are the "real" objects as saved in the interface, and thus show the effects of Manhattanization and polygon decomposition. Again, though, what is displayed on-screen is separate from the data in the interface, so modifying or deleting the objects will not affect the interface. To change geometry, the interface must be cleared, and the new objects saved with the **Save Selected** button or equivalent.

If the technology saved in the "`.sav`" file differs from the current technology, a warning will be issued. New layers will be created as needed, and one can use the interface normally, for this data set. If cleared, however, any new data set will build up a layer sequence from the current technology file, plus any new layers that were created.

Defining FastHenry Terminals

In order to use FastHenry, the contact points between which inductance is measured must be specified. The **RL Terminal Edit** button in the **RLC Extraction** panel places *Xic* in a mode where these terminals can be defined. FastCap does not use terminals.

There are some subtleties in defining terminal locations that if not understood by the user can lead to frustration and confusion. Please read this section thoroughly before attempting to use this feature.

First, terminals must appear in pairs in each conductor group. A conductor group is a set of conducting objects that are connected together by vias. Each conductor group can have exactly two terminals, or exactly zero terminals. Any other condition will generate an error. The editor allows no more than two terminals to be defined in each conductor group.

Second, a terminal is specified by defining a rectangular area. This area must enclose one or more nodes of a conductor. The "nodes" are internal locations where conductors can make contact. This is the main subtlety; the user must know where the nodes are. It is possible to define a terminal box that covers no nodes, which will generate an error.

The nodes are located at points in the following locations:

1. The midpoints of the sides of each partition rectangle.
2. The center of each partition rectangle.

The partition rectangles are the highlighted rectangles as displayed in the FastHenry partition editor, which is described below. As the partitioning is refined, more nodes are created, as the partition boxes are subdivided. Thus, whether or not a terminal box covers any nodes may be affected by the level of partitioning. It does not matter whether terminal boxes are defined before or after the partitioning is refined.

It is highly recommended that terminals be located so as to contact nodes that are defined by the default partitioning, i.e., before the partition editor is used. These will always be valid.

The default partitioning for an object is generated as follows:

1. The via areas are subtracted from the conductor.
2. The conductor region is broken up into rectangles, and projections of every edge of the polygon are used to subdivide every rectangle.
3. The rectangles are subdivided along enclosed edges from other objects.
4. The operation loops through all conductors of all layers, until all subdivision has been done.

The result is a set of rectangles that cover each conductor, that represents the default partitioning. This division is normally invisible, however the partition rectangles can be viewed in the partition editor. It is the midpoint of the sides of each of these rectangles, plus the central point, which are the nodes.

Generally, terminal boxes are defined along the edge of a conducting object. If, for example, the edge is the end of a long strip, then the terminal box can enclose the entire edge, guaranteeing that it will cover the node(s). This is true in general — an edge will always contain at least one node.

It is not a problem and is often desirable that the terminal box enclose more than one node. This may roughly correspond to the difference between a point and a linear contact. All nodes enclosed by a terminal box are made equivalent with FastHenry "`.Equiv`" statements.

Once the **RL Terminal Edit** button is engaged, terminal boxes can be defined, by pressing button 1, dragging over the area, and releasing. One can also press, hold motionless for a brief period, and release. Then, pressing a second time will complete the rectangle.

Terminal boxes are created for the current layer only. An attempt to create a terminal box on a non-conductor layer will result in an error message. A terminal box must intersect an object on the layer. If the box is successfully created, a highlighting box will remain visible. The editor allows no more than two terminals to be defined in each conductor group. Note that the terminal boxes do not snap to the grid.

If a terminal box is defined that intersects an existing box on the same layer, the existing box will be deleted, and no new box is created. One can also simply click on an existing box to delete it.

When defining terminals, the first terminal defined in a conductor group is the reference terminal. This is the terminal that will be listed first in the "`.external`" line in the FastHenry input file. In the on-screen display, this terminal is shown with a diagonal line across the upper left corner. One can think of these as the terminals with the "dot" in the dot convention.

When the interface processes the geometry, which occurs before FastHenry file generation, the primary node number is shown on-screen in the center of each terminal box. The node numbers are generated internally, and the numbers shown are the same as those that appear in the `.external` lines of the Fasthenry file. This makes the correspondence obvious.

It is very important that the current layer be set to the layer of the object on which the terminal is intended. If you forget to change the layer, then the terminal may instead be assigned to an underlying object on a different layer, leading to errors.

FastHenry Partition Editor

As was described in the section discussing terminals, the initial partition divides at all boundaries of the object, and at the boundaries of the objects on conducting layers above and below the object. This initial partitioning is generally not bad, however for greatest accuracy one may wish to further subdivide the partitions in areas with high field gradients. This will increase FastHenry execution time, but if done correctly improves accuracy. Generally, process variation leads to quite a bit of uncertainty anyway, so obtaining extreme accuracy is probably not worth the effort. The FastHenry partition editor provides the capability of refining (subdividing) the partitioning where necessary.

Pressing the **RL Partition Edit** button in the **RLC Extraction** panel starts the FastHenry partition editor. The editor remains active until the **Esc** key is pressed, the **RL Partition Edit** button is pressed again, or some other command is invoked which forces exit.

Initially, all partitions for all objects on all layers are displayed. Clicking on a conducting object will change the display to show only the partitioning for that object, which will be termed the "current object". Clicking on another object will change the display to show the new object's partitioning, and make it the current object.

Clicking on the current object will "refine" the partition where the user clicked. The original partition rectangle will be replaced with nine smaller rectangles that cover the same area. Pressing and dragging over the current object will refine all of the partitions that overlap the rectangle defined by dragging.

The layer-specific mode can be used to limit object selection to specific layers. This is often necessary to ensure that the button presses apply to the current object, rather than selecting another object at the same location (or vice-versa).

It is presently not possible to "un-refine" a partition box. However, pressing the **Delete** key will, for the current object, destroy all partitioning, and recreate the initial partitions.

The partitioning is remembered between invocations of the partition editor, until the interface is cleared, reset, or a new object is saved. The partitioning can be saved with the **Dump Saved** button in the **RLC Extraction** panel.

The interface partitions the conductors into oriented volume elements. The connection points of the volume elements are at "nodes" which are points in space. The nodes for adjacent volume elements must coincide for current to flow.

Since we do not know *a-priori* which direction current is flowing in the x-y plane, volume elements must support flow in both directions. Since the films are thin, the z-direction is ignored. A basic volume element is a rectangular prism which consists of four segments and five nodes. A "segment" is a rectangular parallelopiped oriented between two nodes, that carries current between the nodes. There is a node in the center of each element edge, and one in the center of the element. Two of the segments are x-directed, and two are y-directed. The x and y segments join at the central node. This element is used to tile the conductors. The tiling is such that all tiles that have a neighbor will share a node

with the neighbor. The highlighted rectangles that are visible in the partition editor are the outlines of these volume elements. Given the spatial node coincidence constraint, some thought should reveal why it is necessary to refine an element by trisecting rather than, for example, bisecting. This is necessary to retain the node at the midpoint of each side of the volume element.

FastCap Partition Editor

For FastCap, conductor surfaces and dielectric interface planes are partitioned into rectangles. Each partition rectangle serves as a container for the charge and polarization accumulation for that region. For greatest accuracy, one should use a fine decomposition, at least in regions with high field gradients. However, increasing the number of partition rectangles increases FastCap execution time.

The initial partitioning is applied to the top and bottom surface of all conducting objects, to the "vertical" edges of the conductors and vias, and to the interface planes that separate dielectrics. The "vertical" partitions are created internally and are not editable. Their width is the film thickness, and length is limited by FcEdgeMax. For conductor top and bottom surfaces, the via areas are first subtracted, and the resulting polygons decomposed into boxes. The boxes are split along projected edges from the current object, and along overlapping edges from other objects. Each box is split further if a side is larger than the **FcPartMax** value. Unlike the FastHenry partitioning, there is no constraint on where divisions of the rectangles can occur. For simplicity, the same partitioning is used on the top and bottom of conductors. The space left for a via on the opposite side is automatically filled in. At each outside edge of each rectangle, an additional, thin, edge partition is created. This will have width (normal to the edge) given by the **FcThinEdge** value. The edge partitions account for the greater charge accumulation near edges. Dielectric interfaces are initially partitioned in an analogous manner.

At this point, the partitioning may be quite rough, and FastCap accuracy would be poor, with a high potential for matrix malformation errors. The partitioning may require further refinement to improve accuracy. One way to accomplish this automatically is to set **FcPartMax** to a small value, however this often becomes impractical due to memory and execution time limitations. The FastCap partition editor allows the user to refine the partition only where necessary. The partition panels should be small where field gradients are large. Using the partition editor requires some intuition about E&M, and experimentation will aid the user in developing the art.

When the partition editor starts, the partitioning of all objects is shown. Clicking on a conducting object will change the display to show only the partitioning for that object, which will be termed the "current object". Clicking on another object will change the display to show the new object's partitioning, and make it the current object.

With the current object defined, clicking on or dragging over the partition panels will refine each of those panels into four new panels. The process is repeated to provide sufficiently small panels where necessary.

The layer-specific mode can be used to limit object selection to specific layers. This is often necessary to ensure that the button presses apply to the current object, rather than selecting another object at the same location (or vice-versa).

It is presently not possible to "un-refine" a partition box. However, pressing the **Delete** key will, for the current object, destroy all partitioning, and recreate the initial partitions.

If '**c**' is pressed, cut mode is enabled. A vertical or horizontal line is attached to the pointer. Pressing '**/**' will switch the orientation. Clicking on a panel will divide the panel along the line. Dragging over multiple panels will cut each panel dragged over. Press '**c**' again to exit cut mode.

The interface planes are associated with layers, and not objects. These may also require refinement.

The interface plane partitioning is invisible until the user switches to interface edit mode by pressing the '**i**' key.

The interface that is shown, and can be edited, is determined by the current layer selection in the layer menu. If a conducting layer is selected, the interface plane is the one located at the bottom surface of that layer. Selecting a via layer will display the interface at the top surface of that layer. Selecting the passivation layer will display the interface at the top surface of the passivation. These partitions are refined using the same technique as for conductors. Pressing '**i**' again exits the interface editing mode, and the interface partitioning becomes invisible.

The partitioning is remembered between invocations of the partition editor, until the interface is cleared, reset, or a new object is saved. The partitioning can be saved with the **Dump Saved** button in the **RLC Extraction** panel.

11.18.2 The RLC Extraction Panel

This panel, brought up by the **Extract RLC** button in the **Extract Menu**, controls the interface that allows capacitance to be extracted with the FastCap program, and resistance and inductance to be extracted with the FastHenry program. The interface can also be controlled to a large extent with the **!fx** prompt line command.

At the top of the panel is the **Dataset Name** text entry field. This allows the user to enter a word which is taken as a name for the current set of data in the interface. This word will be used as a base name for files generated with the interface, for input to FastHenry or FastCap, and for other purposes. If no data set name has been entered, the default name used will be "`unnamed`".

To the right is a **Help** button which will bring up text describing this panel.

Below are two columns or entry widgets: text entries on the left, push buttons on the right. The left column entries are the following:

FxUnits
: This is an option menu which is used to set the length units used in FastCap and FastHenry 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 FastCap of FastHenry files are generated.

The text entries in the left column control default sizes assumed in the interface for processing geometry. Each has an associated **!set** variable, which will be set or updated when a non-default entry is given in the text area. If the variable is set with the **!set** command, the text shown in the text area will be updated unless frozen.

Once the interface processes the saved objects, the geometric factors are no longer relevant, and the entry areas are frozen to the values currently in use in the interface. In this state, the entries can not be changed, and setting the associated variables will have no effect. If the interface is reset with the **Reset** button or cleared with the **Clear Saved** button, then the entries revert to the normal behavior.

FxMinRectSize
: This numerical entry field is used to specify the minimum rectangle width or height used when non-Manhattan objects are converted to a Manhattan approximation when they are saved in the interface. The value is entered in microns, and the default value is 1.0. This is tied to the FxMinRectSize variable, which can also be set directly with the **!set** command.

FxPlaneBloat
This numerical entry field specifies the distance outside the bounding box of all conductors that FastCap interface planes and dark-field conductors in FastHenry or FastCap will extend. The value is given in microns, and the default is 10.0. This entry field is tied to the FxPlaneBloat variable, which can also be set with the **!set** command.

FcPartMax
This is used to set the maximum length or width of a FastCap partition panel used on the top of bottom surface of a conductor, or in a dielectric interface. The value is given in microns, with 10.0 being the default. This entry is tied to the FcPartMax variable, which can also be set with the **!set** command.

FcEdgeMax
This numeric value is used to set the maximum length (along the edge) of a "vertical" edge panel used for FastCap. The width of these partition panels is always the layer thickness. The value is given in microns, with 10.0 the default. The entry is tied to the FcEdgeMax variable, which can also be set with the **!set** command.

FcThinEdge
This numerical entry specifies the width (normal to the edge) of the thin panels used along top/bottom conductor outside edges for FastCap. The value is given in microns, with 0.5 being the default. The entry is tied to the FcThinEdge variable, which can also be set with the **!set** command.

Reset
This button will destroy all partitioning and revert the interface to the state just after all objects have been saved. This will un-freeze the text areas listed above,

The right column consists entirely of push-buttons, each of which performs an operation or enters a mode. These are described below.

Save Selections
When pressed, selected objects are deselected, and conducting objects are read into the interface. The label at the bottom of the panel will change to reflect the number of objects that have been saved.

This can be invoked at any time, while in any other command, and will operate on the selections in effect. While in the **Paths** and **Quick Paths** commands, all of the objects in the current path are added to the storage.

If the interface has already been "run", i.e., a file generated or the program executed (the text entries are frozen in this state), then the interface will be reset, as if the **Reset** button was pressed. All partitioning will be destroyed.

Clear Selections
The interface will be cleared of all geometry and partitioning.

Dump Saved
The state of the interface, including the geometry, partitioning, and layer sequencing and parameters, will be saved to a file. The default name for this file is the data set name followed by a "`.sav`" extension, but this can be changed from the dialog that will pop up. This function can be used to save a data set for later use.

Recall Saved
This enables a data set that has been previously saved with **Dump Saved** to be recalled. Any existing information in the interface will be overwritten. If the recalled data set was created with a different technology file, a warning will be issued but any new layers will be created as needed, and the user can continue with any processing required.

C Partition Edit
This puts the interface into FastCap partition editing mode.

Extract C
This button will dump a temporary FastCap input file, run FastCap, and display the results. The result file is named *dataset_name*−*pid*`.fc_log`, where *dataset_name* is the name of the data set, or "`unnamed`" if this is not given, and *pid* is the process id of the spawned process used to run FastCap. The file contains listings of the input file produced by the interface and the output file produced by FastCap.

By default, FastCap 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 FastCap is running, one can continue using *Xic*.

If the FxForeg variable is set with the **!set** command, then FastCap will instead run in the foreground. In this case, the result file is named *dataset_name*`.fc_log`, and *Xic* will be unresponsive until the run completes.

Dump FastCap File
This button allows a FastCap input file to be produced. The default name for this file is *dataset_name*`.lst`, where *dataset_name* is the current data set name, or "`unnamed`" if none was given. This file is a combined list file readable by the version of FastCap distributed by Whiteley Research Inc.

RL Partition Edit
This button engages the partition editor for FastHenry.

RL Terminal Edit
This button enters the FastHenry terminal editing mode.

Extract RL
This button will dump a temporary FastHenry input file, run FastHenry, and display the results. The result file is named *dataset_name*−*pid*`.fh_log`, where *dataset_name* is the name of the data set, or "`unnamed`" if none was given. The *pid* is the process id of the spawned process used to run FastHenry. This file contains listings of the input file produced by the interface, and the output and impedance matrix files produced by FastHenry.

By default, FastHenry 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 FastHenry is running, one can continue using *Xic*.

If the FxForeg variable is set, with the **!set** command, then FastHenry will instead run in the foreground. In this case, the result file is named *dataset_name*`.fh_log`, and *Xic* will be unresponsive until the run completes.

The result file contains the results from the impedance matrix, converted to inductance, using the run frequency. In addition, if there are exactly two ports (four terminals), the "transmission impedance" will be computed for non-dc runs. This is

$$L_1 + L_2 - 2L_{12}$$

and represents the lumped inductance one would have for a transmission line.

Dump FastHenry File
This button allows a FastHenry input file to be produced. The default name for the file is *dataset_name*.`inp`, where *dataset_name* is the current data set name, or "`unnamed`" if none was given. If the Lambda parameter has been applied to any conductor, i.e., the conductor is a superconductor, then the version of FastHenry distributed by Whiteley Research Inc. should be used. Otherwise, the file should be compatible with any version of FastHenry.

There are three additional rows containing text entry areas, located below the two columns of input widgets just described.

FcArgs
This text entry area can be given a string, which will be included in the argument list when FastCap is run with the **Extract C** button. This allows specialized FastCap command line options to be provided during the run, which the user may require. This entry field is tied to the FcArgs variable, which can also be set with the **!set** command.

FhArgs
This text entry area can be given a string, which will be included in the argument list when FastHenry is run with the **Extract RL** button. This allows specialized FastHenry command line options to be provided during the run, which the user may require. This entry field is tied to the FhArgs variable, which can also be set with the **!set** command.

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.

11.19 The Edit Extraction Button: Set Parameters

Pressing the **Edit Extraction** button brings up the **Extraction Parameter Editor**. This includes a text window which contains a list of the the keywords (as described in 11.1.1) and parameters for the extraction subsystem for the current layer. The text can be modified or added to, which will change the keyword definitions for the current layer. Selecting a new current layer will change the text to that appropriate for the new layer. A message will indicate if errors were encountered. Changes made will be reflected in subsequently generated technology files produced with the **Save Tech** command button in the **Attributes Menu**. These keywords apply to layers in physical mode only.

Listed keyword lines can be selected by clicking on them with the mouse. A selected line can be deleted with the **Delete** button in the **Edit** menu, or may be edited with the **Edit** button in the same menu. In **Edit**, the user is prompted for the appropriate values in accord with syntax expected for that keyword line. A modified or deleted line can be recovered with the **Undo** button.

A new keyword line can be added by selecting one of the entries in the **Keywords** menu. The user is prompted for any values associated with the keyword. The addition (or replacement) can be undone with the **Undo** button.

When adding a keyword, redundant and inconsistent keywords that are already in the list, such as a previous instance of the keyword, are removed.

The **Attributes** menu allows extraction-related parameters that do not correspond to layers to be set. At present, there is only one such parameter:

SubstrateEps *diel_const*
: This keyword sets the relative dielectric constant assumed for the substrate, used by the FastCap interface. If not given, the default is 11.9.

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.

The syntax generated for keyword entries is exactly that used in the technology file. Each line contains a keyword plus any arguments. The keywords are described in 11.1.1.

Chapter 12

The User Menu: User Commands and *Xic* Scripts

The **User Menu** contains two built-in commands. Other buttons which appear execute user-generated scripts, or pop up menus of user-generated scripts. An entry will appear for each script found along the script search path. The built-in commands are summarized 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

The **User Menu** supports a mechanism for creating submenus, which contain buttons that launch other scripts or sub-submenus. This can be useful for organizing the user's scripts into categories.

12.1 Script Menus: User-Defined Sub-Menus

Submenus 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);
[nosort]
name1 path1
name2 path2
...
```

The first line must be a CIF comment line in the same format as other library files. The *libname* can be anything and is ignored. For top-level submenus, the basename of the file, stripped of the "`.scm`" suffix, will appear on the button in the **User Menu** which produces the pop-up menu.

Blank lines and lines starting with '`#`' are ignored. If a line contining 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.

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 sub-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-submenu. Each *name* should be unique in the menu, duplicates are ignored. The scripts can be conveniently placed in a subdirectory of the directory containing the script menu file, which is not itself in the script path. If the reference is to another script menu, 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.

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**.

12.2 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 template cell capability.

In addition to the native scripting capability, *Xic* provides an interface to the popular open-source Tcl/Tk scripting language.

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.

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.

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.

The default system wide location for scripts is in the directory
`/usr/local/lib/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 (see 1.2.4), 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 otherwise specified, the search 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 of "`.scr`", function libraries which are named "`library`", and script menu files, which will produce a drop-down submenu in the **User Menu**. Script menu files must have an extension "`.scm`". In addition, auxiliary files such a images, referenced in the help text, may also be present.

Scripts can also be included in the technology file itself. These scripts will appear as buttons in the **User Menu**, as with other scripts. This should be useful for including simple technology-specific commands, such as those that create special extraction layers. However, scripts in the technology file can not be loaded into the debugger.

In the technology file a script is included as follows. The Script keyword is followed by the name which will appear in the command button. This will supersede any script file with the same name. The script text must be terminated with the keyword EndScript. See A.1 for more information.

The **!script** command is yet another means by which scripts can be placed into the **User Menu**. This command associates a keyword, 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 entry word in the **User Menu** is unique in the menu. Script execution has the following priority when names clash:

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.

12.3 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.

12.4 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 determined by which of the **File** or **Execute** drop-down menus was last visited. Opening the **File** menu switches to editing mode. Execution mode is set when the **Execute** menu is opened.

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 **New** button in the **File** menu will clear the previous contents of the text window, allowing a new script to be keyed in. If the previous 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.

The **Save** button in the **File** menu 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.

The **Open** button in the **File** menu 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 open pop-up is active, a script is selected in the **User Menu**, that script name will be loaded into the open dialog text area. Files can also 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.

The **Print** button in the **File** menu brings up a control panel for sending the contents of the text window to a printer, or to a file.

The **Quit** button in the **File** menu 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 **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.

The **Run** command in the **Execute** menu 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** will cause the script to pause at the next line.

The **Step** command in the **Execute** menu causes the current line to be executed, and the current line pointer will be advanced to the next line.

The **Reset** button in the **Execute** menu will reset the current line to the start of the script.

Under Unix/Linux, 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.

The **Monitor** button in the **Execute** menu 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.

12.5 The Rehash Button: Rebuild User menu

This button 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.

12.6 Supplied Example Scripts

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.

fullcursor
: 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.

preferences
: This command executes a script which allows many of the variables which control *Xic* behavior to be set graphically. Otherwise, these variables are usually set with the **!set** command. The variables that are currently set can be listed with "`!set`" (i.e., no arguments), and a list of variable names with meaning to *Xic* is shown for "`!set ?`".

  Due to the sheer number, some variables, particularly those that are really obscure or can be set graphically elsewhere, have been omitted.

spiral
: 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
: 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.

yank
: 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.

This page intentionally left blank.

Chapter 13

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 13.5. Preprocessor directives can be used to comment out blocks of lines.

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.

> **Note**: This behavior was different in release 2.5.35 and before. In those releases, the script parser, when reading the following constructs, would assume that the construct occupies the rest of the current line of text. That is, the parser would stop reading after reading the construct, and anything that follows on the line would be ignored. However, this behavior could be modified if a semicolon (';') followed the construct. In that case, the parser would continue reading after the semicolon, as if reading a new line.
>
> `goto` *label*
> *statement*
> `end`
> `break` [*n*]
> `continue` [*n*]
> `return` [*statement*]
> `function` *xxx*(...)
> `endfunc`
> `static` ...
> `global` ...
>
> In addition "preprocessor" lines (`#ifdef`, `#define`, etc.) and comments have the same property, but any trailing semicolon is ignored.

In the current release, the parser will continue reading after most of these constructs, whether or not a semicolon is present. Thus, 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.

A semicolon will explicitly terminate a construct, and may be needed in rare cases where the parser has trouble interpreting the syntax. The only place where a semicolon is required is to terminate the `static` and `global` declaration lists, if further command text follows on the line.

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 D.1.1. The parser also supports user-defined functions.

13.1 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.

13.2 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.

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.

lexper
: This variable type contains a parse tree for a "layer expression" (see 10.1). 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) and the promotion of scalars to handles when a handle is assigned to a scalar. Variables that are referenced before assignment 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.

13.2.1 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.

13.2.2 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 most other commands, so returns 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*. Note that strings are not copied in assignment, so if several variables point to the same string, they will all see any modifications to the string.

Example:

```
s1 = "a string"
s2 = s1 + 2
Print(s2)        # prints "string"
```

```
s1[4] = ' '
Print(s2)        # prints "st ing"
Print(s1)        # prints "a st ing"
```

13.2.3 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 "fast" 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 from C. Once an array has been defined, subsequent use allows expressions as the index values.

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.

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.

```
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. 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).

13.2.4 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. When through with a handle, it should be closed (with the `Close` function) to free the memory used by the data object. If passed to the `Print` family of functions, a string giving the type of handle will be printed.

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 (13.3) 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 to free internal resources.

13.2.5 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 (13.3) 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.

13.2.6 Lexpers

The lexper 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 lexper is a piece of compiled code that can execute very quickly. Functions that accept a lexper 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 lexper variable.

These variables can not be assigned, and no operators can be applied. They can be passed to functions only.

If passed the the `Print` family of functions, the layer expression string will be printed.

13.3 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
| , `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.

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 $x = x - 1; y = x$

13.3.1 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 beyond the end of the string, however it is possible to define a string pointer to data in memory ahead of the string, which is not a good thing.

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.

Array Overloads

Pointer arithmetic is discussed in the section describing array variables (13.2.3).

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.

13.4 Control Structures

As in C, logical "true" is indicated by a nonzero value. The following control statements are accepted.

13.4.1 `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 13.7), the function returns immediately with the value of the *expression*, if given.

13.4.2 `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).

13.4.3 `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.

13.4.4 `while`

`while` *expression*
    statements
`end`

On each pass through the loop, if *expression* evaluates nonzero, execute *statements*, otherwise exit the loop.

13.4.5 `dowhile`

`dowhile` *expression*
    statements
`end`

On each pass through the loop, execute *statements*, then evaluate *expression*. If *expression* evaluates to zero, exit the loop.

13.4.6 `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
```

13.4.7 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
```

13.4.8 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")
```

13.5 "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 keywords start with the comment '#' character.

There is one predefined macro. For any script which is read from a file (not counting the technology file) the token `THIS_SCRIPT` is effectively defined (as with `#define`, see below) 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")
```

`#define` *token*
`#define` *token*`(`*arg*`, `*arg*`1, ..., `*arg*`n) [`*text_containing_args*`]`

The macro name *token* may use alpha-numeric 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.

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.

`#ifdef` *token*

If *token* has been defined, execution resumes at the following line. Otherwise, execution resumes at the line following the next `#else` or `#endif`.

`#ifndef` *token*

If *token* has not been defined, execution resumes at the following line. Otherwise, execution resumes at the line following the next `#else` or `#endif`.

`#else`

Used in conjunction with `#ifdef` and `#ifndef`.

`#endif`

Used to terminate an `#ifdef`, `#ifndef`, or `#else` block, or a `#macro` block.

`#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.

13.6 Math Functions

The following math functions are defined internally.

Name	Returns
`abs(`x`)`	absolute value of x
`acos(`x`)`	arc-cosine of x
`asin(`x`)`	arc-sine of x
`atan(`x`)`	arc-tangent of x
`atan2(`x`, y)`	arc tangent of x, y
`acosh(`x`)`	arc-hyperbolic cosine of x
`asinh(`x`)`	arc-hyperbolic sine of x
`atanh(`x`)`	arc-hyperbolic tangent of x
`ceil(`x`)`	smallest integer >= x
`cos(`x`)`	cosine of x
`cosh(`x`)`	hyperbolic cosine 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
`ln(`x`)`	natural logarithm of x
`log(`x`)`	base 10 logarithm of x
`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

These functions behave as do 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.

13.7 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 12.2) 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 13.9).

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**.

13.8 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 12.2) 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.

13.9 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.

13.10 Predefined Constants

The following constants are recognized by name:

`e`	Natural log base
`pi`	3.14159...
`PI`	3.14159...
`NULL`	0.0
`INFINITY`	Maximum extent of object coordinate field
`TRUE`	1
`FALSE`	0
`EOF`	-1

The value of `INFINITY` is one million microns. This is the value of "infinity" in the cell coordinate system used by *Xic*.

13.11 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.

13.11.1 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">
```

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>
```

13.11.2 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.

13.12 The Tcl/Tk Interface

Xic has an interface to Tcl/Tk. Tcl (Tool control language) is a popular open-source scripting language, and Tk is a graphical package addition. Since this capability is dynamically loaded, *Xic* can use this capability if it has been installed, but does not require the installation. This interface is presently not available under Microsoft Windows.

If the Tcl/Tk libraries are installed in the standard location (`/usr/local/lib`), the libraries should be found automatically. If these libraries are installed elsewhere, the following variables should be set to indicate the locations to *Xic*.

```
!set TclLibrary path_to_tcl
!set TkLibrary path_to_tk
```

The *path_to...* are the full paths to the Tcl/Tk shared libraries. These variables can be set in a `.xicinit` or `.xicstart` file, or in the technology file.

The **!tk** command is used to execute a Tcl/Tk script. The language syntax is provided in documentation supplied with Tcl/Tk, and is described in

Tcl and the Tk Toolkit, John K. Ousterhout, Addison-Wesley.

The command syntax is

```
!tk script_path arguments...
```

where *script_path* is a path to a file containing the script, and any arguments follow. 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 functions for *Xic* scripts are available in Tcl/Tk in the form

`xic` *function arguments...*

The function `xic` is a Tcl function which loads the interface function or user-defined function given in the second argument. 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. The variable type of the argument is inferred from the argument:

- 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 grey *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**.

13.13 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
```

This page intentionally left blank.

Chapter 14

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 (`xterm`) window in which the user can issue operating system commands. When the user is finished, typing "`exit`" will close the window. 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. The shell which is used to execute operating system commands can be selected by the user, through the Shell variable and the **!set** command.

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: !#
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.

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.

Conversions	
!assemble	Merge archive files
!sced2xic	Convert from sced format
!gzip	Apply compression to file
!gunzip	Uncompress a file
Current Directory	
!cd	Change working directory
!pwd	Print working directory
Diagnostics	
!logfiles	Browse the log files
!compare	Compare geometry in files
!empties	Check for empty symbols
!checkgrid	Check object for off-grid vertices
!checkover	Report cells that overlap
!dups	Select coincident identical objects
!check	Check electrical input for consistency
!regen	Regenerate damaged file
!polycheck	Check polygon characteristics
!polymanh	Select Manhattan polygons
!polyfix	Fix polygon errors
!polynum	Show polygon vertex indices
!bincnt	Database diagnostic
!setflag	Set cell flags
!summary	Print summary info of current hierarchy
!fileinfo	Print info about archive file in console
!time	Print elapsed run time in seconds in console
!xdepth	Print transform stack depth in console
Design Rule Checking	
!showz	Show DRC partitioning
!errs	Rebuild errors database
!errlayer	Create error polygons on some layer
Extraction	
!source	Update schematic from SPICE file
!exset	Update schematic from physical data
!addcells	Add missing cells
!find	Find devices
!ptrms	Physical terminal manipulations
!terms	Update electrical terminals
!ushow	Show unassociated elements
!fx	Control FastCap/FastHenry interface
!fxcell	Create cell from FastCap/FastHenry interface
Fonts	
!dumpfont	Dump label font to file
!dumplogofont	Dump logo font to file
!logofont	Set font for **logo** command
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
Layers	
!layer	Create logical layers
!ltab	Manipulate layer table
!lal	Manipulate layer alias table
!ltsort	Alphanumerically sort layer table
Libraries	
!mklib	Create or append to a library file
Memory Management	
!clearall	Clear all memory
!vmem	Windows only, print virtual memory statistics
!mmrecycle	Assert or vacate recycle mode
!mmclear	Clear recycle mode free lists
!mmstats	print memory manager statistics
Operations	
!mo	Move objects
!co	Copy objects
!ro	Rotate objects
!rename	Rename subcells
!sa	Save current cell
!sqdump	Save selections as native cell
!cont	Read contents of native cell
!svq	Save selections in register
!rcq	Recall selections from register
Physical	
!box2poly	Convert boxes to polygons
!path2poly	Convert wire paths to polygons
!poly2path	Convert polygon boundaries to wires
!bloat	Expand/contract object
!area	Measure object area
!join	Join objects into polygon
!split	Split polygon into trapezoids
!manh	Convert to Manhattan polygons
!polyrev	Reverse polygon winding
!noacute	Eliminate acute angles
!togrid	Move selected object vertices to grid
!tospot	Condition object for spot size
!bb	Print bounding box of current cell
!origin	Set origin of current cell
!import	Import structures into the current cell
Rulers	

!dr	Delete rulers
Scripts	
!script	Add a script to the **User Menu**
!exec	Execute a script
!lisp	Execute a Lisp script
!tk	Execute a Tcl/Tk script
!listfuncs	Pop-up list of saved functions
!rehash	Re-read script libraries and rebuild **User Menu**
!rmfunc	Delete a saved function
Selections	
!select	Select objects
!desel	Deselect objects
!zs	Zoom to selected objects
Symbol Table	
!symtab	Manipulate symbol tables
Variables	
!set	Set/examine variables
!unset	Unset variables
Window Cells	
!cc	Load cell into window
!sw	Swap cells in main and subwindow

14.1 Conversions

14.1.1 The !assemble Command: Merge Archives

Syntax: `!assemble` *specfile*

This command implements a feature whereby several archives can be merged into a larger archive, on-the-fly so as to avoid memory limitations. The output consists of a definition of a top-level cell, containing instances of cells found in the archives, plus the translated records from the archive files. The input and output can be any of the supported archive formats, in any combination. The operation is controlled by a specification file which supplies the file names and the description of the top-level cell. It is a simple text file, prepared by the user, containing a number of keywords with values. The format is illustrated below.

Only physical data are translated, electrical data will be stripped. A log file s produced when the **!assemble** command is run. This is named "`assemble.log`" and is written in the current directory.

The file used for output must have a recognized extension that corresponds to the file type. 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. Input file types are recognized by content, so there is no name requirement.

Below is a description of a specification file. Blank lines are ignored, comment lines begin with the '#' character. Each active line begins with a keyword, from the list below. Keywords are case-insensitive.

`OutFile` *out_file_name*
: This is mandatory, and provides the name of the output file.

`TopCell` *cellname*
: Optional top-level cell added for placing subcells found in archives.

`Source` *filename*
: Name of a source file, of which there must be one or more. This starts a block of lines that apply to that source. For best (or sometimes even acceptable) performance, the input file should *not* be compressed if any `Place` lines (see below) are associated.

The following are the "source options". If these appear ahead of the first `Source` line, then they set defaults that apply to all source blocks. If they appear following a `Source` line, they apply to that source only and override any defaults.

`OnlyLayers` [*list_of_layer_names*]
: Use only the layers listed. If no *list_of_layer_names* is found, the list that applies is taken from the list provided by any of `OnlyLayers`, `SkipLayers`, or `LayerList` currently in scope.

`SkipLayers` [*list_of_layer_names*]
: Skip the layers in the list. If no *list_of_layer_names* is found, the list that applies is taken from the list provided by any of `OnlyLayers`, `SkipLayers`, or `LayerList` currently in scope.

`LayerList` *list_of_layer_names*
: Set the list of layers to be used with `OnlyLayers` or `SkipLayers`. This is implied when a *list_of_layer_names* is provided with these keywords.

`LayerAliases` *name1=alias1 name2=alias2 ...*
: List of layer aliases to apply.

`ConvertScale` *scale_factor*
: Scaling to apply, 0.001 through 1000.0.

`ToLower`
: Flag to indicate conversion of upper case cell names to lower case.

`ToUpper`
: Flag to indicate conversion of lower case cell names to upper case.

`CellNamePrefix` *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*
: Cell name change suffix. This operation occurs after case conversion. The *suffix_string* is interpreted in the manner of the InCellNameSuffix variable.

This concludes the source options.

`Place` *cellname* [*placement_name*]
: Each source block can have zero or more instance blocks. An instance block starts with a `Place` line as shown. The hierarchy under *cellname*, which must name a cell in the source file, will be extracted and added to the `TopCell`, if given, or streamed if not. The *placement_name*, if given, will replace *cellname* in output. In either case, any cell name alteration will be applied. The actual names used in output will be internally altered if necessary so that each output cell name is unique, by adding "$*N*", where *N* is an integer. If a 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. If no `TopCell`, there would be no addition to output.

`PlaceTop` [*placement_name*]
: The `PlaceTop` line is equivalent to a `Place` line, except that it will automatically select the first top-level cell found in the source. It is equivalent to the `Place` line with the name of this cell as the first argument. This is convenient when the cell name is unknown.

The following are the "instance options". If these appear ahead of the first `Source` line, the option sets a default which is applied to all instances in all source blocks. If these appear after a `Source` line but ahead of the first `Place` line in the block, the option sets a default which applies to all instances in the source block, overriding any defaults. If these appear following a `Place` line, the option applies to that instance only and overrides all defaults. These will be ignored in source blocks that contain no instances.

`Translate` *x y*
: Specify the translation coordinates. If not given, default is 0,0.

`Rotate` *angle*
: Specify a rotation angle, must be a multiple of 45 degrees. If not given, default is no rotation.

`Magnify` *magn*
: Specify a magnification. If not given, default is 1.0.

`Reflect`
: Apply mirror-Y transformation (before rotation, if any).

`NoEmpties`
: If given, empty subcells under *cellname* will not be written to output. Layer filtering will often produce empty cells.

`Flatten`
: If given, all geometry under the cell being placed will be written as part of the cell being placed, i.e., the cell hierarchy will be flattened.

`Window` *left bottom right top*
: If given, only the subcells and objects needed to describe the given area in the cell being placed will be written. The coordinates apply to *cellname* before any scaling is applied, and are given in microns.

`Clip`
: If `Window` was given, this will cause geometry to be clipped to the window.

14.1.2 The !sced2xic Command: Convert Old Format

Syntax: `!sced2xic`

This command provides an update path for files generated using the sced feature of the `Jspice3` program. When **!sced2xic** is given in the prompt area, the current cell will be converted from the Jspice3 sced format to *Xic* format. To use, open the sced cell for editing. It will be a mess. Then issue the **!sced2xic** command. The devices will be moved to correct (or at least better) locations, and labels will be added. Some editing may be necessary since the relative positions of device terminals may not be the same. The cell can then be saved as an *Xic* cell. This must be repeated with the subcells (one can use **Push**/**Pop**). It should not be done more than once, or to a cell that doesn't need it. The command can be undone.

14.1.3 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 (except for the older Solaris distribution) 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*.

14.1.4 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 (except for the older Solaris distribution) 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*.

14.2 Current Directory

14.2.1 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.

14.2.2 The !pwd Command: Print Directory

Syntax: `!pwd`

This command will print the *Xic* current working directory on the prompt line.

14.3 Diagnostics

14.3.1 The !logfiles Command: Browse Log Files

Syntax: `!logfiles`

This command 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.

14.3.2 The !compare Command: Compare Hierarchies

Syntax: `!compare [-l` *layer_list*`] [-t` *obj_types*`] [-s] [-d] [-e] [-g] [-r` *max_errors*`]` *file1* `[`*file2*`] [-c` *cell* `...]`

This function compares the geometry and instance placements in two cell hierarchies, one of which is in an archive file and the other is in memory or in another archive file. The results are written to a log file.

Options:

`-l` *layer_list*
: The *layer_list* is a 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*.

`-t` *obj_types*
: The *obj_types* ia 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.

  When `-g` is given (see below) only '`c`' is recognized, "`bpw`" is assumed. That is, when using geometric comparison, '`c`' will control whether or not to compare subcell instances, and boxes, polygons, and wires will always be tested (labels are ignored).

`-d`
: Don't record the actual differences, only whether or not the cells differ.

`-e`
: Expand subcell arrays (if comparing subcells). Cell arrays are converted to individual placements before comparison, avoiding false errors between arrayed and equivalent unarrayed layouts.

`-g`
: If set, the objects on the comparison layer are converted to a trapezoid list, and geometric operations on the trapezoid lists are used for the comparison. Labels are ignored in this mode, and subcells are treated normally. If not set, objects are compared directly, so differences will be recorded even if the resulting geometry is the same, e.g., cells with and without **!split** will show many differences. Subcells are always checked directly, thus an array vs. individual placements will indicate differences.

`-r` *max_errors*
: The integer *max_errors* is the maximum number of error objects returned, per cell and per layer or instance symbol. The value 0 (the default) implies no limit.

`-c` *cell...*
This option must appear last in the command line. The tokens following `-c` are cell names. If these are given, only the listed cells will be considered, otherwise all cells will be considered.

There must be at least one archive file name given. If a second archive file name is found, comparison is between the two hierarchies. Otherwise, the second hierarchy is expected to exist in memory.

Only the cells for the current mode (electrical or physical) are considered. Properties are ignored.

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 `-c` is given.

When the comparison finishes, the user is given to option to view the results.

14.3.3 The !empties Command: Check for Empty Symbols

Syntax: `!empties`

This command will search through the hierarchy rooted in the current cell for empty symbols. An empty symbol is a cell which contains no physical or electrical objects or subcells. This test is performed automatically when a new cell is opened for editing, though this can be suppressed by setting the NoCheckEmpties variable.

If empty cells are found, a pop-up appears, which allows the user to delete the empty symbols. The pop-up is similar in appearance to the **Merge Control** pop-up, which appears when a cell in memory may be overwritten. 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.

If all the listed cells are deleted, *Xic* will search again for empty cells, and update the pop-up if any are found. Additional cells may now be empty due to the deletions.

14.3.4 The !checkgrid Command: Mark Off-Grid Vertices

Syntax: `!checkgrid [c][o]`

This 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 will be tested, or all objects on the current layer if in layer-specific mode. In either case, 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.

14.3.5 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.

14.3.6 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.

14.3.7 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 `convert_rd`*XXX*`.lib` file produced when a cell is read, if repairs were made as the cell was read.

The functionality of the **!terms** command is included in the **!check** command.

14.3.8 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.

14.3.9 The !polycheck Command: Check Polygon

Syntax: `!polycheck` [*layer ...*]

This will apply the reentrancy test on all polygons (no arguments) or polygons on the layers given in the current cell. Polygons that do not pass are selected.

For certain conditions, **!polycheck** will repair the error silently. The following conditions are repairable:

- 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.

14.3.10 The !polymanh Command: Select Manhattan Polygon

Syntax: `!polymanh` [*arg*]

Without an argument, this command will deselect all polygons, and then select only those that are Manhattan, on the current layer only if in layer-specific mode (all layers otherwise). If there is an argument, which can be any text token, the non-Manhattan polygons will be selected instead.

14.3.11 The !polyfix Command: Fix Polygon

Syntax: `!polyfix`

This command will remove duplicate and in-line redundant vertices from selected polygons.

14.3.12 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.

14.3.13 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.

14.3.14 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. You should probably never have (or want) to use this. The second form of the command brings up a window containing a list of the flag names and descriptions.

14.3.15 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.

14.3.16 The !fileinfo Command: Show File Statistics

Syntax: `!fileinfo` *filename* [*flags*] [*outfile*]

This will print to the console window information about the archive file given as the first argument. The optional second argument is an integer whose bits represent the types of information to return. See the description of the `CxInfo` script function for detailed information on these flags and the corresponding output.

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.

14.3.17 The !time Command: Show Elapsed Time

Syntax: `!time`

Print the elapsed program run time, in seconds, in the console window.

14.3.18 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 1. 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.

14.4 Design Rule Checking

14.4.1 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.

14.4.2 The !errs Command: Regenerate DRC Error Markers

Syntax: `!errs`

This command can be called when the DRC **Show Errors** command is active. It deletes any error indicators, and recreates the errors from the current error file. This is particularly useful after a **Check In Background** operation as the error indicators are not normally created. The **!errs** command makes all the errors visible, as with the **Check In Foreground** command. The indicators can be cleared with the DRC **Clear Errors** command.

14.4.3 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 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.

14.5 Extraction

14.5.1 The !source Command: Source SPICE File

Syntax: `!source` [-[n]a] [-[n]c] [-[n]f] *spice_file*

This command is the prompt-line equivalent of the **Source SPICE** button in the **Extract Menu**. It enables a schematic to be created or updated from a SPICE file.

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.

There are three internal mode switches: all devs, create, and clear. These correspond to the three check boxes in the pop-up from the **Source SPICE** command.

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.

clear
: If set, the electrical part of a cell is cleared before reading the SPICE input. This implies create.

The default behavior of the command can be changed by setting the following boolean variables with the **!set** command. These determine the states of the buttons in the **Source SPICE** pop-up.

SourceAllDevs
SourceCreate
SourceClear

The options in the command line can be used to override the mode set by the current state of these variables.

-a	all devs on	-c	create on	-f	clear on
-na	all devs off	-nc	create off	-nf	clear off

14.5.2 The !exset Command: Extract Parameters and Set

Syntax: `!exset` [-[n]a] [-[n]c] [-[n]f] [-[n]w] [*depth*]

This command is the prompt-line equivalent of the **Source Physical** button in the **Extract Menu**. This will generate and update the properties of electrical devices from the parameters extracted for the corresponding physical device. All devices are acted on, to *depth* in the hierarchy of the current cell. If *depth* is not specified or given as 0, only the current cell is updated. Devices can be updated individually with the **Compare Devices** button in the **Extract menu**.

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.

There are four internal mode switches: all devs, create, clear, and include wire cap. These correspond to the check boxes in the pop-up from the **Source Physical** command.

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.

The default behavior of the command can be changed by setting the following boolean variables with the **!set** command. These determine the states of the buttons in the **Source Physical** pop-up (the logic is inverted for variables with a "No" prefix).

NoExsetAllDevs
NoExsetCreate
ExsetClear
ExsetIncludeWireCap

The options in the command line can be used to override the mode set by the current state of these variables.

-a	all devs on	-c	create on	-f	clear on	-w	include wire cap on
-na	all devs off	-nc	create off	-nf	clear off	-nw	include wire cap off

14.5.3 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 **Edit Menu**).

14.5.4 The !find Command: Find Devices

Syntax: `!find` [*devicename*[.*prefix*[.*index*]]]

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, and the *index* is the instance number. Periods are used to separate the fields, any of which can be empty. This command will draw a highlighting box around each matching device in the physical layout, draw a box around each contact of the device, and print the device name and contact names. With no argument, the highlighting is erased. This command is useful for testing the device extraction capability.

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.

14.5.5 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.

14.5.6 The !terms Command: Regenerate Terminals

Syntax: `!terms`

This command is obsolete. It is actually run automatically when a cell is loaded for editing. It is also called within the **!check** command. If the terminals of a cell were updated, a message stating this will be added to the `convert_rd`*XXX*`.log` file which is produced during the read operation.

Earlier versions of *Xic* did not include physical terminals, thus files from these versions don't directly support the features which use these terminals, such as LVS and node naming. This command can be used to update an old file. This will create any missing terminals from the terminals specified in the device specifications in the device library file. If the device library file is also old, the terminals will have to be added by hand with a text editor, or a newer device library file should be used. Terminals are added only in the current cell, so the process must be repeated in any subcells.

Newer versions of *Xic* use the terminal structures to store the names of the terminals of subcircuits. It is important that the cells contain a terminal structure for each node, or certain commands or actions may not work correctly.

A cell suspected of having "old-style" nodes (without terminal structures) can be dealt with as follows. An old-style cell may work perfectly well for generating SPICE output, but there may be problems at some point (for example, the **Dump Elec Netlist** command won't work for formats other than SPICE), since the internal data are incomplete. To check if a cell has old-style nodes: Open the cell for editing, and press the **subct** button in the electrical mode side menu. If the cell is OK, then the white box subcircuit terminal markers should appear. If they don't, press the **Info** button in the **View Menu**, then press **Shift** and click anywhere in the drawing window. The **Info** window will contain info about the cell. Near the bottom of the listing, where the properties are listed, look for lines like "`10 7 1 -2000 29000`", i.e., properties that start with 10. If there are any, the cell has old-style nodes.

To fix, type "`!terms`". That's it. The **subct** command should bring up the white markers, and a *new* **Info** window should list the same property as (for example) "`10 7 1 -2000 29000 _1 0 0 8 ?`".

14.5.7 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 side menu button or otherwise. Mostly, the objects are simply selected, however objects such as physical devices use other highlighting code.

14.5.8 The !fx Command: Control FastCap/FastHenry Interface

Syntax: `!fx` *keyword* [*arg*]

This command is a prompt-line equivalent to most of the functionality of the FastCap/FastHenry interface. This interface is also controlled from the **RLC Extraction** panel, which is produced by the **Extract RLC** button in the **Extract Menu**.

The first argument is a keyword, which must be present and must be one of those listed below. The second argument is optional or required for some of the keywords. Most of the keywords indicate an operation that is equivalent to pressing one of the buttons in the **RLC Extraction** panel.

`save`
: This will save the currently selected objects on conducting layers into the FastCap/FastHenry interface. This is equivalent to pressing the **Save Selections** button.

`clear`
: This will reset and clear the interface, and is equivalent to pressing the **Clear Saved** button.

`reset`
: This will destroy existing partitioning in the interface and recreate the initial partitioning, equivalent to pressing the **Reset** button.

`dataset` [*dataset_name*]
: This will apply the name provided in the second argument as the data set name. If no name is given, the data set name is cleared, meaning that "`unnamed`" will be used. This is equivalent to the **Dataset Name** text field in the panel.

`setref` *cellname*
: This will set the "reference cell". The reference cell is a flat cell in memory containing objects identical to those saved in the preprocessor, and is used to supply group numbers which will be used in FastCap output. When no reference cell is defined, the preprocessor generates conductor numbers.

`area` *left*,*bottom right*,*top*]
: This sets the minimum computation area, and should be called before objects are added. The computation area is otherwise the bounding box of all conductors added.

`dump` [*filename*]
: This will dump the interface context to a file, equivalent to pressing the **Dump Saved** button. If a second argument is given, it will be used as the file name, otherwise the default name is used.

`recall` *filename*
This will recall a saved data set into the interface, from the file given in the second argument. This is equivalent to the **Recall Saved** button.

`partc`
This will start FastCap partition editing mode, equivalent to pressing the **C Partition Edit** button.

`parth`
This will start FastHenry partition editing mode, equivalent to pressing the **RL Partition Edit** button.

`term`
This will start FastHenry terminal editing mode, equivalent to pressing the **RL Terminal Edit** button.

`exit`
This will terminate any of the editing modes, as if the **Esc** key was pressed.

`dumpc` [`-s`] [*filename*]
This will dump a FastCap input file using the name given in the argument, or the default name if no name is given. This is equivalent to the **Dump FastCap File** button.

If the optional "`-s`" option is given, this will append to or create two files: *filename*`.top` and *filename*`.bot`. If the files exist, they will be appended to. Concatenating the two files (e.g. `cat` *file*`.top` *file*`.bot` > *file*`.lst`) will produce a standard unified list file. This option enables output from different FastCap preprocessor runs to be combined.

`dumph` [*filename*]
This will dump a FastHenry input file using the name given in the second argument, or the default name if no name is given. This is equivalent to the **Dump FastHenry File** button.

`runc`
This will initiate a FastCap run, equivalent to pressing the **Extract C** button.

`runh`
This will initiate a FastHenry run, equivalent to pressing the **Extract RL** button.

14.5.9 The !fxcell Command: Create Cell From RLC Extraction Interface

Syntax: `!fxcell` [*cellname*]

This command will create a cell from the contents of the FastCap/FastHenry interface, and make it the current cell. This is similar to the behavior after pressing **Recall Saved** in the **RLC Extraction** panel, and as in that case the "real" Manhattanized geometry is shown. If an argument is given, it will be used as the cell name, otherwise the name is "`$$`*dataset_name*`$$`", where *dataset_name* is the name of the data set, or "`unnamed`" if none was given. Any existing cell in memory with the same name will be overwritten.

14.6 Fonts

14.6.1 The !dumpfont Command: Dump Vector Font File

Syntax: `!dumpfont` [*filename*]

This command will dump the current font used for labels to *filename*, or to `xic_font` if *filename* is not given. Any existing file with the same name will be backed up with a `.bak` extension. A font file found along the library search path will be read on program startup. The description of the **label** command in 4.11 contains information on the file format.

14.6.2 The !dumplogofont Command: Dump Logo Font File

Syntax: `!dumplogofont` [*filename*]

This command will dump the current font used for physical text (created with the **logo** command) to *filename*, or to `xic_logofont` if *filename* is not given. Any existing file with the same name will be backed up with a `.bak` extension. A font file found along the library search path will be read on program startup. The description of the **label** command in 4.11 contains information on the file format.

14.6.3 The !logofont Command: Set Logo Font

Syntax: `!logofont`

This command brings up the **Font Selection** pop-up allowing the user to select a font for use in the **logo** command. When the **Set Pretty Font** button is pressed, the name of the chosen font (if any) is saved in the LogoPrettyFont variable. If the **Pretty** button is pressed, the LogoPretty variable is set, and the LogoManhattan variable is unset. Thus, the **logo** command will use the "pretty" font. Pressing the **Manhattan** button will set the LogoManhattan variable and unset the LogoPretty variable, and the **logo** command will use the internal Manhattan font. Pressing the **Internal** button unsets the LogoPretty and LogoManhattan variables, thus the **logo** command will use the vector font.

14.7 Graphics

14.7.1 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.1.7). 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.

14.7.2 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.

14.8 Grid

14.8.1 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 **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.

14.8.2 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.

14.9 Help

14.9.1 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.

Information on the help database is provided in A.7. 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://www.wrcad.com`" is correct, giving only "`www.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.

14.9.2 The !helpfont Command: Set Help Font

Syntax: `!helpfont` *fontfamily-size*

This specifies the default font family used in HTML viewer (help) windows. This is the font, which is generally proportional-spaced, used to render most text. Under X-windows, this is the X Library Font Descriptor font family name with "-*size*" appended, where *size* is the base pixel size. Under Microsoft Windows, this command does nothing.

If no argument is given, the default font family and size, which is

```
adobe-times-normal-p-14
```

is reset. Otherwise the argument is expected to be a font family specifier in the format described.

This command has limited value, as the fonts are most conveniently set with the **Set Font** button in the help viewer.

14.9.3 The !helpfixed Command: Set Help Fixed Font

Syntax: `!helpfixed` *fontfamily-size*

This specifies the default fixed font family used in HTML viewer (help) windows. The fixed font is used to render typewriter and preformatted text. Under X-windows, this is the X Library Font Descriptor font family name with "-*size*" appended, where *size* is the base pixel size. Under Microsoft Windows, this command does nothing.

If no argument is given, the default font family and size, which is

```
misc-fixed-normal-c-14
```

is reset. Otherwise the argument is expected to be a font family specifier in the format described.

This command has limited value, as the fonts are most conveniently set with the **Set Font** button in the help viewer.

14.9.4 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.

14.10 Layers

14.10.1 The !layer Command: Generate Layers

Syntax: `!layer [join|split|splitv]` *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. 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.

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 LayerPartSize 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.

When joining objects, there are several variables which fine-tune the operation. See the description of the **!join** command (14.14.6) for information.

If *layer_name* does not exist in the physical 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 10.1). 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.

The assigned-to layer, if it exists, is cleared before the operation. The command is optionally undoable. An additional prompt asks the user if the operation is to be undoable. If yes, memory usage will be larger and processing time longer. The **!layer** command prompts for a hierarchy depth. If the depth given is greater than 0, there is a second prompt: "Recursively create layer in subcells?". If the

response is 'y', the new layer is created in subcells to the given depth. If the response is 'n', the layer is created only in the current cell, but uses geometry from the subcells up to the given depth, in effect producing a flattened layer.

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
```

Advanced Features

The layer names in layer expressions in the **!layer** command can be specified as *layer_name*[.*cellname*]. The *layer_name* is an existing layer name. If the name of a cell follows a '.' separator, then the geometry is obtained from the given cell, as if it were overlaid on the current cell. Either of the *layer_name* and *cellname* can be double quoted, and must be quoted if the name contains a '.' character, for example `"some.long.name"."mycell.xic"`.

The *cellname* can have the form [$*symtab*$][*cellname*], i.e., the name of the cell can be preceded by a dollar-sign delimited string token, which is the name of a named symbol table. If 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 cell name is optional, and if not given defaults to the name of the current cell.

If the $*symtab*$ is given, and the cell is not in this table, it will be opened from disk into the given table (not the current table).

This provides another means of importing geometry from another cell, after performing the manipulations of the expression. 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.

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:

```
!symtab old
edi oldstuff/mycell
!symtab
edi newstuff/mycell
```

There are two versions of `mycell` in memory ("`edi`" is the accelerator for the **Edit** command). 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.

If the **!layer** command is used on a large, complex cell, it may take some time. There is presently no "working" feedback.

14.10.2 The !ltab Command: Modify Layer Table

Syntax:

`!ltab add` *layername ...*
`!ltab remove` *layername ...*
`!ltab rename` *oldname newname*

This command has three forms, corresponding to the keyword given as the first argument.

If the second word is the literal `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.

The second form removes the listed layers from the layer table. These can be reinserted from the **Layer Editor** in the **Attributes Menu**. Removed layers can also be added to the end of the layer listing by typing "`!layer` *name*" or with the "`!ltab add`" form of this command.

The third form renames the layer named *oldname* to *newname*.

14.10.3 The !lal Command: Manipulate Layer Alias Table

Syntax: `!lal` *keyword* [*arguments*]

This command allows manipulation of the layer alias table. The keyword and arguments are:

`read` *filename*
The *filename* is a path to a file containing layer alias definitions. Each line of the file contains a single definition, in 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-byte hex number, using upper case, where the left two bytes represent the hex value of the layer number, zero padded, and the right two bytes represent the zero padded datatype number.

`dump` *filename*
This will print the contents of the layer alias table to the file whose path is given as an argument. The format is as described above. The filename can be "`stdout`" or "`stderr`", in which case output will go to one of these channels, rather than to a file.

`clear`
The layer alias table will be cleared.

`add` *name newname*
This will add *newname* as an alias for *name* to the layer alias table. If *name* is already aliased, it will not be updated, and a warning will appear (remove it first to change the alias).

`remove` *name ...*
Any alias definition for *name* will be removed from the layer alias table. Additional names can appear, and they will be removed as well.

`print` *name ...*
This will print on the prompt line the aliases associated with each name given.

Layer aliasing is made active by setting the UseLayerAlias variable. Similar manipulations are available from script functions.

14.10.4 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.

14.11 Libraries

14.11.1 The !mklib Command: Create Library File

Syntax: `!mklib [`*archive_file*`] [-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 `-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.

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.

14.12 Memory Management

14.12.1 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.

14.12.2 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.

14.12.3 The !mmrecycle Command: Set/Clear Recycle Mode

Syntax: `!mmrecycle [y|n]`

There is a memory manager mode, where for certain frequently used list elements, the elements are never freed but are instead "recycled" when needed. This should improve performance for certain operations such as geometry computation, at the cost of keeping memory in use that can potentially be freed (but can be truly freed later with the **!mmclear** command).

Without an argument, the recycling mode is toggled, otherwise the mode is set according to the "`y`" or "`n`" argument. Switching recycling off does not free the free lists, use **!mmclear** to reclaim this memory for other uses.

14.12.4 The !mmclear Command: Clear Recycle Free Lists

Syntax: `!mmclear`

This will free the free lists that are saved in recycling mode (as set with the **!mmrecycle** command), giving the memory back to the system. This is implicitly called by **!clearall** and the `ClearAll` script function.

14.12.5 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.

14.13 Operations

14.13.1 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. If in layer-specific mode, objects being moved that are on the current layer will be moved to *layer_name*. If not in layer-specific mode, all objects moved will be changed to *layer_name*. Subcells are moved without regard to *layer_name*.

There is a companion **!co** (copy) command.

14.13.2 The !co Command: Copy Objects

Syntax: `!co` *x* [*y* [*layer_name*]]

The **!co** command will copy 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 copy. It should be the name of a layer that is not the current layer. If in layer-specific mode, objects being copied that are on the current layer will be copied to *layer_name*. If not in layer-specific mode, all objects copied will be placed on *layer_name*. Subcells are copied without regard to *layer_name*.

There is also a companion **!mo** (move) command.

14.13.3 The !ro Command: Rotate Objects

Syntax: `!ro` *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 fourth 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. If in layer-specific mode, objects being rotated that are on the current layer will be moved to *layer_name*. If not in layer-specific mode, all objects rotated will be changed to *layer_name*. Subcells are rotated without regard to *layer_name*.

14.13.4 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.

14.13.5 The !sa Command: Save Cells

Syntax: `!sa`

Invoking this command is the same as invoking the **Save** command in the **File Menu**.

14.13.6 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.

14.13.7 The !cont Command: Read Contents of Native Cell

Syntax: `!cont` *cellpath*

The *cellpath* is a path to a native symbol file, or the name of a cell in memory. The cell will be read if necessary, and the contents of the cell will be ghost-drawn and "attached" to the mouse pointer. The objects will be placed in the current drawing where the user clicks. This is equivalent to placing the cell and flattening.

14.13.8 The !svq Command: Save Selections in Register

Syntax: `!svq` [*regnum*]

This will save the current selections into a "register" which can be recalled later. 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.

14.13.9 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.

14.14 Physical

14.14.1 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.

14.14.2 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.

14.14.3 The !poly2path Command: Polygon to Outline Conversion

Syntax: `!poly2path`

This will convert selected polygons to wires. The wires have zero width, and the first and last points are the same. The wires follow the polygon boundary.

14.14.4 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 in layer-specific mode, only selected objects on the current layer are acted on, otherwise all selected objects are acted on. If the *dimen* is positive, the parts of edges that do not contact or overlap with an 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.

Presently, there are four bloating algorithms available, selectable by an optional numeric second argument. If this argument is missing or 0, the default algorithm will be used.

Only boxes, wires and polygons are affected. Wires and boxes become polygons after the operation. An object is deselected if it is modified.

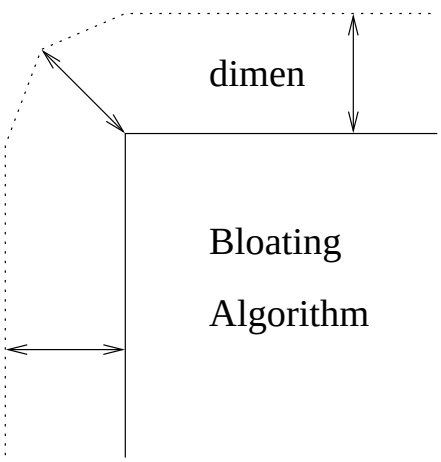
mode = 0 (default)

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 14.1 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.

mode = 1

Figure 14.1: The default algorithm used in the **!bloat** command to enlarge an object.



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.

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.

mode = 3

This is very similar to mode 0, however it uses the DRC sizing functions (this was used for mode 0 in earlier releases).

This algorithm 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.

14.14.5 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.

14.14.6 The !join Command: Join Adjacent Objects

Syntax: `!join`

This will combine selected touching objects on the same layer into polygons. This can reduce the size of the data file.

If in layer-specific mode, only selected objects on the current layer are joined. Otherwise, all selected objects will be joined with selected objects on the same layer.

The **!join** command, and 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.

To join a set of objects, the first step 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 1000.

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 100.

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 trapezoids may have complicated boundaries that interleave the polygons 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.

14.14.7 The !split Command: Atomize Objects

Syntax: `!split` [v|V|1]

This is basically the reverse of **!join**. Selected polygons and wires will be converted to collections of boxes and four-sided polygons.

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. Otherwise, splitting favors the horizontal orientation.

14.14.8 The !manh Command: Convert to Manhattan Polygons

Syntax: `!manh` *min_box_size*

This command applies to selected polygons, on the current layer only if layer-specific mode is in effect (all layers otherwise), and will convert each polygon to a Manhattan approximation. The argument is the size, in microns, of the minimum box width/height used to approximate non-Manhattan parts of the polygon.

The operation works by first decomposing the polygon into horizontal trapezoids, each of which is divided horizontally if necessary so that it can be further split into rectangular and triangular pieces. The triangular pieces are approximated with rectangles using *min_box_size*, then a new polygon is produced from all of the rectangular pieces. This results is a Manhattan polygon that may be very different from that achieved by, for example, a gridding operation.

14.14.9 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.

14.14.10 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.

14.14.11 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.

14.14.12 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. Typically, this size is 0.1 to 0.5 microns, 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. *Xic* has features to precondition round objects to avoid this problem: the SpotSize variable and the **!tospot** command.

The **!tospot** command will apply an algorithm (described below) to all selected polygons. This will apply to any polygon. The *spotsize*, if given, is the spot size to use in milli-microns. Values of 1 – 1000 are accepted. If not given, the value is taken from the SpotSize variable. If this is not set or set to zero, a value will be prompted for.

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.

14.14.13 The !bb Command: Print Bounding Box

Syntax: `!bb`

In physical mode, this prints the bounding box coordinates of the current cell, in microns.

14.14.14 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.

14.14.15 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.

14.15 Rulers

14.15.1 The !dr Command: Delete Rulers

Syntax: `!dr` [*arg*]

This will delete currently displayed rulers, as generated by the **Ruler** 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.

14.16 Scripts

14.16.1 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.

14.16.2 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 submenu 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.

14.16.3 The !lisp Command: Execute Lisp Script

Syntax: `!lisp` *filename* [*args ...*]

PRELIMINARY
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.

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.

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 2.7).

A node name that can't be resolved will generate an error.

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) $\Longleftrightarrow x\ \|\|\ y$
colon	colon(x y) $\Longleftrightarrow' (xy) \Longleftrightarrow x : y$
setq	setq(x y) $\longleftarrow x = y$
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

14.16.4 The !tk Command: Execute Tcl/Tk Script

Syntax: `!tk` *scriptfile*

This command will execute a Tcl/Tk script (see 13.12), contained in the file given as an argument. The Tck/Tk interface is not available under Microsoft Windows. This command will dynamically load the Tcl/Tk libraries, if they are installed. If the Tcl/Tk libraries are installed in the standard location (`/usr/local/lib`), the libraries should be found automatically. If these libraries are installed elsewhere, the following variables should be set to indicate the locations to *Xic*.

```
!set TclLibrary path_to_tcl
!set TkLibrary path_to_tk
```

The *path_to...* are the full paths to the shared libraries. These variables can be set in a `.xicinit` or

`.xicstart` initialization file, or in the technology file.

The `!tk` command is used to execute a Tcl/Tk script. The language syntax is provided in documentation supplied with Tcl/Tk, and is described in

Tcl and the Tk Toolkit, John K. Ousterhout, Addison-Wesley

Additional information can be found at `www.scriptics.com`.

14.16.5 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.

14.16.6 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**.

14.16.7 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**.

14.17 Selections

14.17.1 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*`]`

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 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 ('$') forces matching at the end of a string. Thus, to match a literal, one should use the form "^*string*$".

14.17.2 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.

14.17.3 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.

14.18 Symbol Table

14.18.1 The !symtab Command: Set Symbol Table

Syntax: `!symtab [`*name*`]`

This command is the interface to an important capability: named symbol tables. The symbol table is the internal data structure that maps the names of cells in memory to the cell's data. By default, there is only one symbol table, and all opened cells are kept in the same "space". This command is used to open new symbol tables, or to switch between symbol tables. The *name* can be any text word, and it identifies a symbol table. If *name* is not given, the symbol table used will be the default symbol table. If *name* is "?", a list of the symbol tables presently defined is shown on the prompt line.

When switching between symbol tables, the user is prompted to save any unsaved cells. After the symbol table is switched, the default "noname" cell is opened for editing, in the new symbol table. The **Cells Listing** pop-up will list the cells that have been opened in the current symbol table.

This feature allows cells of the same name to be in the database simultaneously.

Only the default symbol table is checked for unsaved cells when the program exits. It is intended that the named symbol tables not be used for editing, but for reference purposes.

If there are different versions of the same cell name in different symbol tables, they should be written to disk under different names or they will overwrite one another.

14.19 Variables

14.19.1 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 C. 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.

The **preferences** script, found in the **User Menu**, provides a graphical interface to the internal variables normally set with the **!set** 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 1.2.3). 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 1.2.4) 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.

14.19.2 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.

14.20 Window Cells

14.20.1 The !cc Command: Load Cell into Window

Syntax: `!cc [`*name*`] [`*cellname*`]`

This command will change the cell displayed in the drawing window that had the keyboard focus when this command was initiated. If the focus was the main window, this command is similar to the **Edit**

command, and will open a new cell or file for editing. Otherwise, the given file/cell is opened and displayed in the window.

The first argument is the name of or path to an archive file, native cell file, or cell in memory. The second argument is the name of a cell in the archive, used to resolve ambiguity. These are the same arguments as given to the **Edit** command, and this command has the same behavior when reading the layout data as the **Edit** command.

If no argument is given and the focus was a subwindow (from the **Viewport** command), the subwindow will display the same cell as shown in the main window, thus enabling editing in that subwindow if the display mode is the same as the main window.

If no argument is given and the focus was the main window, the cell shown in the first subwindow will be shown in the main window and become the current cell. The subwindows are examined in order of the window number 1-4 as shown in the title bar. If there are no subwindows present or the first window is already showing the current cell, there is no effect.

14.20.2 The !sw Command: Swap Cells in Main and Subwindow

Syntax: `!sw [`*name*`] [`*cellname*`]`

This command must be initiated when a subwindow (as created with the **Viewport** command) has the keyboard focus. It will swap the cells, display modes, and views between the subwindow and the main window, thus making the cell displayed in the subwindow the current cell, allowing it to be modified.

Appendix A

File Formats

The following sections describe in detail the various file formats used by *Xic*.

A.1 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 expected to be found along the library search path, which can be set with the environment variable XIC_LIB_PATH. The default path is

```
( .  /usr/local/lib/xictools/xic/startup ).
```

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, then possibly some 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, followed by script definitions, and user-defined design rules.

The layer blocks follow, which is where the core information about the particular technology resides. The electrical layers are defined first, followed by the physical layers. These are optionally followed by the Device blocks, where physical characteristics for device extraction are given. Finally, there is a section containing display attribute specifiers and the hard copy driver parameters.

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 pair near the top of the file:

`Technology` *name*
: The *name* can be any character token (no white space allowed) and gives a name to the technology. This is not used by *Xic*, but is part of the information available for output in scripts and elsewhere, should it be useful to the user.

`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`".

`ReadCds` *filename* PRELIMINARY
: This is part of the Cadence^{TM} compatibility package (see 2.7). The *filename* is the name of or path to a design resource or text-mode technology file. The full path should be given unless the file is in the library search path. This keyword should appear twice, first for the design resource file, and again for the (Cadence) technology file, if these are separate files. This will define the physical layers and attributes such as colors and fill patterns. Layers defined in the present *Xic* technology file will be in addition to these.

A.1.1 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. Note that comments can also be included in the technology file after the '`#`' character, 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.1.2 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. A macro definition, which can appear anywhere in the technology file, follows the keyword "`Define`". The macro name 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.

The macro name search is case dependent, unlike most other keywords in the technology file.

Syntax:

`Define` *name* [*text*]
`Define` *name*(*arg*, *arg1*, ..., *arg*N) [*text_containing_args*]

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
```

A different type of macro is defined using the `Set` keyword, where the words following are parsed into three tokens "*name* `=` *value*". A macro is referred to by `$(`*name*`)`, which is replaced by *value* as the file is read. The variable must be set before being referenced. Neither the *name* nor *value* tokens can contain the character '`)`' or 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, respectively. Substitution is performed once only, non-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.

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
...
LayerName 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)
...
LayerName 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.

Macros defined with the `Define` keyword can be used to alter the sequence of lines read from the technology file. The technology file parser understands the following keywords:

`IfDef` *token*
`IfnDef` *token*
`Else`
`Endif`

The *token* is tested for existence against the names of macros defined with the `Define` keyword. If the token following `IfDef` is found, the lines up to the matching `Else` or `Endif` are read, otherwise they are ignored. `IfnDef` has the reverse logic. The `Else` line (if present) creates, along with the matching `Endif`, a block which is read or ignored with opposite logic to the initial block (`IfDef`/`IfnDef` to `Else`). This feature allows the technology file to be easily reconfigured by simply changing a `Define` at the top of the file. The constructs can be nested to arbitrary depth.

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.1.3 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.

A.1.4 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 1.2.4 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 symbol files and/or libraries. The current directory "." should always be listed first The symbol 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/lib/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 A.7). 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/lib/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/lib/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.1.5 Technology File Scripts

Scripts can be included in the technology file. These scripts will appear as buttons in the **User Menu**, as with other scripts, however scripts defined in the technology file can not be loaded into the debugger. This feature is useful for including simple technology-specific commands, such as those that create special extraction layers or physical features.

A script is included In the technology file as follows. The `Script` keyword is followed by the name which will appear in the command button. This will supersede any script file with the same name. The lines of the script follow, and the script text must be terminated with the keyword `EndScript` on a separate line.

`Script` *script_name*
script text
...
`EndScript`

A.1.6 Technology File Layer Blocks

There is separate specification of layers used for schematic layout and those used for physical layout. The layer used for schematics is always called "SCED", and if not defined in the technology file, it will be created by *Xic*. It is always the first layer, and can not be deleted. Additional layers are used for visual purposes, and for storage of modifications that can be reused later by converting them to the SCED layer.

Each layer definition starts with the keyword `LayerName` for physical layers or `ElecLayer` for electrical layers, followed by a name. In *Xic*, layers have a name, which follows the CIF format (1-4 alphanumeric characters), and an optional "long name" which can be just about any string. In *Xic* commands and functions, either name can be used to access a layer (so the names had better be unique!). If the name string following the keyword does not fit the CIF restrictions, the following actions are taken:

- If the given name is null, empty, or space-only, the new name will be "`null`".
- If the given name is too long, only the first four characters will be used in the name.
- Characters that are not alphanumeric will be replaced with '`X`'.
- If the name is truncated or characters are replaced, the long name will be set to the given name.

The layer specification further consists of a list of keywords in the format given below.

In the table that follows, 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, if '`0`' (zero), '`n`', or '`N`' appear, '`n`' is understood, otherwise '`y`' is assumed.

All of the keywords below are optional, and can appear under an electrical or physical layer, unless stated otherwise. It is not strictly necessary to define any layers at all. The electrical layers, if any, are defined first, followed by the physical layers (electrical and physical layers can not be mixed). A "layer block" consists of the keyword which defines the layer name and all of the following keywords up to the next layer name specification, or end of the file. The opening layer name specification defines the "current" layer, and the keywords that follow apply to that layer.

`ElecLayer` *name*
: This keyword specifies the beginning of the layer block for the electrical layer *name*. Layers will appear in the electrical mode layer table in the order given. The name of the first electrical layer can be given as anything, but will be renamed "SCED".

`LayerName` *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. This keyword should not appear until all of the electrical layers have been defined.

`LongName` *longname*
: This will set (or reset) the long name 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 argument.

`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.

`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 found in the X `rgb.txt` file (this
  - Three space-separated numbers, each 0–255, representing the red, green, and blue intensity. E.g., "`196 240 235`". works for Windows, too). These names can be listed from the **Set Color** pop-up in the **Attributes** menu with the **Colors** button.
  - (Unix/Linux only) 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. Default: 127 127 127

`AltRGB` *colorspec*

This keyword sets the layer color used in hard copy mode, and in the plots on a color device. During hard copy mode (the **Print Control Panel** from the **Print** button in the **File Menu** is visible) these colors can be changed with the **Set Color** command in the **Attributes Menu**, and the new `AltRGB` colors will be set in a technology file created with the **Save Tech** command. If not set, the `AltRGB` colors default to the `RGB` colors.

`Filled y/n`
`Filled` *bit_data* `[y]`

This keyword sets the fill style used to render objects on the layer on-screen. If the first form is used with '`y`', then a solid fill will be used. If '`n`', the figures will be outlined only (empty fill). The second form is used to specify a stipple pattern to use for fill. The pattern data can optionally be followed by '`y`' or '`o`' which indicates that the object should also be outlined. Any additional characters are ignored. The hex code should be zero padded so that each block corresponds to the number of X bytes in the bitmap. The blocks are separated by white space. Thus, for an 8x8 map, there would be 8 blocks of two characters. For 16x16, there would be 16 blocks of 4 characters. The upper left bit of the map is the MSB of the of the least significant byte of the first block. The lower right bit of the map is the LSB or the most significant byte of the last block.
Default: `n`

Examples:
```
filled y
filled cc aa cc aa cc aa cc aa outline
```

Default: n

In electrical mode, the SCED layer defaults to solid fill, and other layers default to empty fill. All layers default to empty fill in physical mode.

`AltFilled y/n`
`AltFilled` *bit_data* `[y]`

This keyword sets the fill style used when hard copy mode is active, and is used in the hard copy plots.
Default: value of `Filled`

`HPGLfilled` *filltype* [*option1 option2*]

This keyword is used only by the HP-GL hard copy driver, 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 applies only to the xfig hard copy driver, 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.

`WpFilled` *filltype*

This keyword applies only to the WordPerfect hard copy driver, and allows one of the internal fill patterns to be associated with the layer. The *filltype* number can be 0–63 or 255. Giving 255 is like empty fill (0), but obscures anything below.

The fill patterns, from the WordPerfect documentation, are:

```
0       Hollow
1       Solid
2       Narrow spaced +45 degree lines
3       Medium spaced +45 degree lines
4       Widely spaced +45 degree lines
5       Narrow spaced +45 and -45 degree lines
6       Medium spaced +45 and -45 degree lines
7       Widely spaced +45 and -45 degree lines
8       Narrow spaced vertical lines (shaded)
9       Medium spaced vertical lines
10      Widely spaced vertical lines
```

```
11      Interspaced dots or density (least dense)
12      Dots
13      Dots
14      Dots (medium density)
15      Dots
16      Dots
17      Dots (most dense)
18      Narrow spaced horizontal lines
19      Medium spaced horizontal lines
20      Widely spaced horizontal lines
21      Narrow spaced vertical and horizontal lines
22      Medium spaced vertical and horizontal lines
23      Widely spaced vertical and horizontal lines
24      Narrow spaced -45 degree lines
25      Medium spaced -45 degree lines
26      Widely spaced -45 degree lines
27      Partial weave
28      Weave
29      Fine checker board
30      Zigzag (horizontal)
31      45 degree wide line
32      Wide checker board
33      Fine 45 degree bricks
34      Medium 45 degree bricks
35      Wide 45 degree bricks
36      Fine 45 degree bricks
37      Medium 45 degree bricks
38      W 45 degree bricks
39      Horizontal brick thin
40      Horizontal brick medium
41      Vertical brick thin
42      Vertical brick medium
43      Waves
44      Wide vertical zigzag
45      Medium vertical zigzag
46      Movie pattern
47      Skewed brick pattern
48      Interconnected dot pattern
49      Shadow squares horizontal
50      Dot black vertical
51      Vertical slanted thick boxes
52      Vertical cursive black
53      45 degree shingle pattern
54      45 degree wide black arrows
55      Horizontal "T" lines
56      Black box 45 degree dots
57      Corner piece pattern
58      45 degree triangles and dot lines
59      Horizontal heartbeat monitor pattern
60      Vertical lines/horizontal waves
61      Light and dark wave patterns
```

```
62      45 degree fill color umbrella pattern
63      45 degree fill color umbrella pattern
255     Use background color for solid fill
```

`Invisible`
If this keyword appears, the layer will not be visible, though it will appear in the layer table, where the visibility status can be changed.
Default: visible

`AltInvisible`
This has the same function as the `Invisible` keyword, but applies to hard copies and to the screen while in hard copy mode.
Default: status of `Invisible`

`Blink`
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

`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

The following keywords set the layer mapping for GDSII and OASIS format input and output. These can be programmed from within *Xic* with the **Edit Parameters** command in the **Convert 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 255, and the datatype can take values -1 through 255. 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 set the mapping of GDSII layers and datatypes to *Xic* layers when a GDSII or OASIS file is read. The format is described in 9.11.

`StreamOut` *out_layer* `[`*out_datatype*`]`
This keyword sets the mapping of *Xic* layers to GDSII layers and datatypes when a GDSII or OASIS file is written. The format is described in 9.11.

`NoDrcDatatype` *datatype*
If this keyword is given, then any object that has the given *datatype* will be ignored during design rule checking. The format is described in 9.11.

The following keywords are used to set design rules in the physical layers. These keywords can appear only in physical layer blocks. See the description of the design rules in 10.2 for more information.

`Connected [Region` *source_expr*`]` [*string*]

`NoHoles [Region` *source_expr*`]` [*string*]

`Overlap [Region` *source_expr*`]` *layer_expression* [*string*]

`IfOverlap [Region` *source_expr*`]` *layer_expression* [*string*]

`NoOverlap [Region` *source_expr*`]` *layer_expression* [*string*]

`AnyOverlap [Region` *source_expr*`]` *layer_expression* [*string*]

`PartOverlap [Region` *source_expr*`]` *layer_expression* [*string*]

`AnyNoOverlap [Region` *source_expr*`]` *layer_expression* [*string*]

`MinEdgeLength [Region` *source_expr*`]` *layer_expr length* [*string*]

`MinArea [Region` *source_expr*`]` *area* [*string*]

`MaxArea [Region` *source_expr*`]` *area* [*string*]

`MinWidth [Region` *source_expr*`]` *width_in_microns* [*string*]

`MinSpace [Region` *source_expr*`]` *space_in_microns* [*string*]

`MinSpaceTo [Region` *source_expr*`]` *layer_expr dimension_in_microns* [*string*]

`MinSpaceFrom [Region` *source_expr*`]` *layer_expr dimension_in_microns* [*string*]

`MinOverlap [Region` *source_expr*`]` *layer_expr dimension_in_microns* [*string*]

`MinNoOverlap [Region` *source_expr*`]` *layer_expr dimension_in_microns* [*string*]

The following keywords, which can appear in physical layer fields, set various flags and variables used for netlist and parameter extraction. These can be programmed from within *Xic* with the **Extraction Parameter Editor** in the **Extract Menu**. See the chapter on extraction (11.1.1) for more information.

`Conductor [ Exclude` *expression* `]`
This keyword indicates that the present layer is to be included in extracted conductor groups.

`Routing`
This keyword implies that the layer is a conductor used for connecting between cells.

Only one of the following two ground plane keywords can appear in the technology file.

`GroundPlane [Global]`
`GroundPlaneDark [Global]` (alias)
This keyword indicates that the present layer is to be treated as a clear-field ground plane.

`GroundPlaneClear [MultiNet]`
`TermDefault [MultiNet]` (alias)
This keyword indicates that the present layer is to be treated as a dark-field ground plane.

`Via` *layer1 layer2* [*expression*]
This keyword indicates that the present layer may provide connection points between conductor nets on *layer1* and *layer2*.

`Contact` *layer* [*expression*]
This keyword defines a layer that may be in contact with another `Conductor` layer, and is to be grouped accordingly in the wire net extraction.

`DarkField`
This keyword indicates that the layer polarity on the chip is the reverse of that shown on-screen.

`Symbolic`
If this keyword is present in a layer block, the layer will *not* be included in the cross section displays produced with the **Cross Section** command (in the **View Menu**).

The next three keywords support electrical parameter extraction for the **Compute Params** command.

`Resis[tance]` *ohm_per_square*
This enables computation of the resistance value of resistor devices.

`Cap[acitance]` *units_per_sqmicron* [*units_per_micron*]
This enables computation of the capacitance of a conductor group.

`Tran[line]` *linethick linepenet grndthick grndpenet dielthick reldielconst*
This keyword will invoke a microstrip model which computes the transmission line parameters along the long dimension of a newly created box.

The following keywords are used by the FastCap/Fasthenry extraction interface. Each keyword takes a single real value argument.

`Thickness` *thickness*
This keyword supplies the film thickness of the corresponding deposited film. The *thickness* is given in microns.

`Rho` *resistivity*
This keyword supplies the resistivity, in MKS units, of the corresponding conducting film.

`Sigma` *conductivity*
This keyword supplies the conductivity, in MKS units, of the corresponding conducting film.

`EpsRel` *diel_constant*
This keyword supplies the relative dielectric constant of insulating layers.

`Lambda` *pene_depth*
This keyword specifies the London penetration depth of superconducting conductors, in microns.

A.1.7 Technology File Attributes

The following keywords appear after the layer specifications, and control various global attributes of *Xic*.

`F`*N*`Key` *text*
The keyboard function keys, usually labeled F1 – F12, can be mapped to the name of a menu button. Pressing that function key is then equivalent to pressing the named menu button. The *text* is the internal name for the command, which is generally a short mnemonic of five characters or fewer. These names are displayed in the pop-up "tooltip" which appears when the mouse pointer

is positioned over a command button, after a short delay. These names 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 menu containing the named button must be active 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*. Note that another program or the window manager may redirect function key presses. It may be necessary to disable these other mappings to use the function keys with *Xic*.

`SubstrateEps` *diel_const*
This keyword sets the relative dielectric constant assumed for the substrate, used by the FastCap interface.
Default: 11.9

`Drc` y/n
This sets whether or not the interactive rule checking is applied to objects being added to the database, controlling the initial status of the **Enable Interactive** button in the **DRC Menu**.
Default: `y`

`DrcMaxErrors` *num*
This keyword sets the maximum number of design rule errors reported in batch mode, at which point checking terminates. If *num* is set to zero, all errors are reported.
Default: 0

`DrcInterMaxObjs` *num*
In interactive design rule checking, this keyword provides a limit on the number of objects checked, to minimize the pause after an operation. If *num* is set to 0, there is no limiting.
Default: 0

`DrcInterMaxTime` *num*
This keyword 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.
Default: 0

`DrcInterSkipInst` y/n
If a subcell is copied, moved, or placed, by default the subcell is tested for design rule violations if in interactive mode. Setting this value to "`y`" will cause this checking to be skipped.
Default: `n`

`DrcNoPopup` y/n
This keyword determines whether errors generated in interactive DRC will be listed in a pop-up window. If set, the messages will not pop up automatically.
Default: `n`

`MergeOnRead` y/n
This keyword determines whether boxes are merged as files are being read.
Default: `n`

`InToLower` y/n
If set to '`y`', symbol names in archive files that are entirely upper case will be mapped to lower case symbol names in *Xic* as the archive file is read. Mixed-case symbol names will not be changed. If set to '`n`' or not present, no such mapping is performed. This will set the initial state of the InToLower variable, which is part of a more general facility for cell name mapping (see 9.1).

For backwards compatibility, an alias `GdsInToLower` is recognized for this keyword.
Default: `n`

`OutToUpper` y/n
If set to '`y`', *Xic* symbol names will be mapped to upper case as an archive file is generated from memory. This sets the initial state of the OutToUpper variable, which is part of a more general facility for cell name mapping (see 9.1).

For backwards compatibility, an alias `GdsOutToUpper` is recognized for this keyword.
Default: n

`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.

`GridSpacing` *spacing*
The *spacing* is a floating point number which represents the default grid spacing (in microns) in both physical and electrical modes.
Default: 1.0

`ElecGridSpacing` *spacing*
`PhysGridSpacing` *spacing*
These keywords can be used to independently set the grid spacing in electrical and physical modes. The last read `GridSpacing` directive will have precedence for a given mode.

`Snapping` *num*
This keyword sets the snapping grid in both physical and electrical modes. The *num* is a positive or negative integer with absolute value of one through ten. If positive, *num* represents the number of snap points per grid interval. If negative, *num* represents the number of grid lines per snap line.
Default: 1

`ElecSnapping` *num*
`PhysSnapping` *num*
These keywords allow the snapping to be set independently for electrical and physical modes. The last `Snapping` directive has precedence for a given mode.

`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*
This sets the style of grid to use in both electrical and physical modes. The style is an 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.
Default: cc (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.

`GridReg`*N resol snap linestyle*
The keywords `GridReg0` – `GridReg7` can be used in the technology file to define the contents of the grid registers. Except for register 0, which is volatile, if a register has been set, the keyword will appear in the output when a new technology file is generated with the **Save Tech** command in the **Attributes Menu**. The *resol* is the grid spacing in microns, *snap* is the snapping number (-10 – 10 excluding 0), and *linestyle* is the line style code.

`Constrain45` y/n
When '`y`' is given, vertices entered to new polygons and wires will be constrained to form angles at multiples of 45 degrees with existing vertices. The rotations in the **spin** command are restricted to multiples of 45 degrees.
Default: `n`

`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

`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`

`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`

`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

`ElecLabelAllInstances` *num*
`PhysLabelAllInstances` *num*
These forms allow the display of unexpanded instance text for electrical and physical modes to be set separately.

`ShowInstanceMark` y/n
If '`y`' is provided, the origin point of selected instances will be marked with a cross in physical mode.
Default: `n`

`ShowContext` y/n
When given '`y`', the context surrounding a subcell is shown during a sub-edit initiated with the **Push** command in the **Edit Menu**. This applies to both electrical and physical modes.
Default: `y`

`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`

`ElecShowTinyBB` *num*
`PhysShowTinyBB` *num*
These forms allow the tiny subcell rendering for electrical and physical modes to be set separately.

`RoundFlashSides` *sides*
This keyword specifies the number of sides to use in the round objects created. The *sides* must be between 8 and 150.
Default: 20

`BoxLineStyle` *style*
This sets the linestyle of the boxes used in electrical mode, and in physical mode for some highlighting purposes such as zooming with button 3. The style is an integer whose binary value is replicated to form the lines used in the box.
Default: e38 (hex)

`Font1-6` *name_of_font*
The `Font1` through `Font6` variables set the default fonts used in the graphical interface. These correspond to the fonts settable from the Font Selection pop-up from the **Set Font** button in the **Attributes Menu**. These are:

`Font1` (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.
Defaults:
Unix/Linux gtk1 default: "`fixed`"
Unix/Linux gtk2 default: "`Monospace 12`"
Windows default: "`Lucida Console 10`"

`Font2` (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.
Defaults:
Unix/Linux gtk1 default: "`-adobe-helvetica-medium-r-normal-*-12-*-*-*-*-*-*-1`"
Unix/Linux gtk2 default: "`Sans 12`"
Windows default: "`MS Sans Serif 10`"

`Font3` (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 gtk1 default: "`fixed`"
Unix/Linux gtk2 default: "`Monospace 12`"
Windows default: "`Lucida Console 12`"

`Font4` (Text Editor Font)
This is the font used in the **Text Editor** pop-up.
Defaults:
Unix/Linux gtk1 default: "`fixed`"
Unix/Linux gtk2 default: "`Monospace 12`"
Windows default: "`Lucida Console 10`"

`Font5` (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 gtk1 default: "`-adobe-times-medium-r-normal-*-14-*-*-*-*-*-*-1`"
Unix/Linux gtk2 default: "`Sans 12`"
Windows default: not used, ignored

`Font6` (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 gtk1 default: "`-adobe-courier-medium-r-normal-*-14-*-*-*-*-*-*-1`"
Unix/Linux gtk2 default: "`Monospace 12`"
Windows default: not used, ignored

In addition, the following two deprecated font keywords are recognized:

`FixedFontName` *name_of_font*
This is the name of a fixed-pitch font that is used in the text areas of the pop-up windows (same as `Font1`).

`FontName` *name_of_font*
This is the name of a font that is used by the text editor pop-up (same as `Font4`).

The *name_of_font* is very different between the Windows and Unix/Linux releases:

Unix/Linux
For gtk1 releases, the name is the X Logical Font Descriptor name for a font available on the user's system, or an alias. For gtk2 releases, the name is a Pango font description name. There is a very modest attempt to interpret a specification of the wrong type.

Windows
The name is in one or the following formats:

New standard (*Xic* release 2.5.52 and later)
face_name pixel_height

Example: `Lucida Console 12`

Old standard (deprecated)
(*pixel_height*)*face_name*

Example: `(12)Lucida Console`

The *face_name* is the name of a font family installed on the system, and the *pixel_height* is the on-screen size.

The user need not worry about this, except that when moving a technology file to Windows from Unix or vice-versa, the fonts may not be as expected. The default fonts should be generally suitable anyway. Fonts are normally set with the **Set Font** command in the **Attributes Menu** and the **Font** button in the **Options** menu of the text editor.

The following keywords set colors used on-screen. In these definitions, the *colorspec* string is the name of a color or an RGB triple:

- The name of a color found in the X `rgb.txt` file (this works for Windows, too). These 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`".
- (Unix/Linux only) 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.

`Background` *colorspec*
This sets the background color used in the drawing windows in both physical and electrical modes. The background can also be set with the **Set Color** pop-up.
Default: 0 0 0

`ElecBackground` *colorspec*
`PhysBackground` *colorspec*
These keywords allow setting of the background colors independently for physical and electrical modes.

A single internal data structure maintains all other attribute (non-layer) colors. All attribute colors can be set from a resource file (Unix/Linux only), as well as from the technology file. Within *Xic*, all attribute colors can be changed with the **!setcolor** command, and from the **Set Color** pop-up.

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 "`Alt`" colors are those used by default while in hard-copy mode. 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*.

Keyword Alias	Default
Prompt Line Colors	
PromptTextColor PromptText	sienna
PromptEditTextColor PromptEditText	black
PromptHighlightColor PromptHighlight	red
PromptCursorColor PromptCursor	blue
PromptBackgroundColor PromptBackground	antique white
PromptEditBackgColor PromptEditBackg PromptEditBackground	white
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
AltCoarseGridColor AltCoarseGrid	CoarseGridColor
AltElecCoarseGridColor AltElecCoarseGrid ElecAltCoarseGrid ElecAltCoarseGridColor	ElecCoarseGridColor
AltPhysCoarseGridColor AltPhysCoarseGrid PhysAltCoarseGrid PhysAltCoarseGridColor	PhysCoarseGridColor
FineGridColor FineGrid	royal blue
ElecFineGridColor ElecFineGrid	FineGridColor
PhysFineGridColor PhysFineGrid	FineGridColor
AltFineGridColor AltFineGrid	FineGridColor
AltElecFineGridColor AltElecFineGrid ElecAltFineGrid ElecAltFineGridColor	ElecFineGridColor
AltPhysFineGridColor AltPhysFineGrid PhysAltFineGrid PhysAltFineGridColor	PhysFineGridColor
Plot Mark Colors	
Color2	red
Color3	lime green
Color4	blue
Color5	orange
Color6	magenta
Color7	turquoise
Color8	sienna
Color9	grey
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

The following keywords set alternative parameters that are used in hard copy mode (when the **Print** button in the **File Menu** is active).

`AltDriver` *name*
This keyword enables the hard copy driver with keyword *name* to be the default hard copy driver in both physical and electrical modes.

`AltElecDriver` *name*
`AltPhysDriver` *name*
These keywords set the default hard copy driver independently for physical and electrical modes. The last `AltDriver` directive has precedence in a given mode.

`AltResolution` *resol*
`AltElecResolution` *resol*
`AltPhysResolution` *resol*
These keywords are obsolete and are no longer recognized. Resolution can be set with the driver block keyword `HardCopyDefResol`.

`AltAxes [Plain | Mark | None]`
This determines the presentation style for the axes in physical mode while hard copy mode is active. 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.

`AltGridSpacing` *spacing*
`AltElecGridSpacing` *spacing*
`AltPhysGridSpacing` *spacing*
The *spacing* is a floating point number which represents the default grid spacing (in microns) in hard-copy mode. These are analogous to the `GridSpacing` keywords.
Default: corresponding `GridSpacing` value

`AltShowGrid` y/n
`AltElecShowGrid` y/n
`AltPhysShowGrid` y/n
These keywords are analogous to the `ShowGrid` keywords, however they determine the grid visibility in hard copies and on-screen when hard copy mode is active.
Default: corresponding `ShowGrid` value

`AltGridOnBottom` y/n
`AltElecGridOnBottom` y/n
`AltPhysGridOnBottom` y/n
These keywords are analogous to the `GridOnBottom` keywords, however they determine whether the grid is shown above or below rendered objects while in hard copy mode.
Default: corresponding `GridOnBottom` value

`AltGridStyle` *style*
`AltElecGridStyle` *style*
`AltPhysGridStyle` *style*
These keywords are analogous to the `GridStyle` keywords, but set the grid used in hard copies and on-screen when hard copy mode is active.
Default: corresponding `GridStyle` value

`AltBackground` *r g b*
`AltPhysBackground` *r g b*
`AltElecBackground` *r g b*
These keywords set the background color when hard copy mode is active, in the same manner

as `Background`, `PhysBackground`, and `ElecBackground`. Presently, most hard copy drivers ignore this specification, and set the background to "clear". The **Color** pop-up will change these colors when in hard copy mode.

A.1.8 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 te be absent; internal defaults will be used.

The driver parameters are specified as keyword-value pairs. The following are recognized:

`HardCopyDevice` *device_name* [off]
Specifies that the following parameters are for 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
versatec_monochrome
versatec_color
wp
wpm
windows_native
xfig_line_draw_color
imlib
```

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** (5.5.2) for more information on these drivers.

`LayerName` *name*
`RGB` *colorspec*
`Filled` ...
`Invisible`
It is possible to define layer presentation attributes specific to individual hardcopy drivers, and to suppress the use of the "`Alt`" attributes for individual drivers. The listed keywords are allowed in the driver blocks. The format of each line is the same as would appear in the layer blocks, however the specification is only used while in hard copy mode, and the driver containing the specification is active. The `RGB`, `Filled`, and `Invisible` lines can only follow a `LayerName` line, and the `LayerName` line must specify a layer already named in a layer block. The name can be that of a physical or electrical layer.

When in hard copy mode with a driver containing layer specifications, and one changes the fill pattern, color, or visibility status of a layer specified in the driver block, the layer specification in the driver block will be updated, and not the "`Alt`" layer. Creating a new technology file with the **Save Tech** command in the **Attributes Menu** will reflect the changes.

In order to specialize layer presentation to drivers, the `LayerName` (at least) line must be added to the technology file with a text editor. Once included, the other presentation attributes can be changed from within *Xic*.

`NoAlt`
The driver blocks can contain this keyword which will cause the use of the ordinary screen presentation while that driver is active in hard copy mode. There is no text following this keyword. If there are layers referenced in the driver block with `LayerName`, their attributes will override the screen attributes.

`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.

`HardCopyPixWidth` *integer_width*
This sets the bitmap width expected by the device, for drivers with a fixed width (Versatec only).

`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. Some drivers may ignore one or more of these values, for example the offsets are ignored by the WordPerfect driver, and the `Yoff` offset is ignored by the Versatec driver. 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.1.9 Example Technology File

Below is the default scalable-CMOS technology file, which illustrates the use of many of keywords and constructs described in this Appendix.

```
Comment Xic technology file for MOSIS scalable CMOS
Comment Whiteley Research Inc., dated 9/23/98 (device blocks added)
Comment Whiteley Research Inc., dated 10/6/98 (device blocks modified)
Comment Whiteley Research Inc., dated 3/15/99 (updated for 2.3.0)
Comment Whiteley Research Inc., dated 3/28/99 (updated for 2.3.1)
Comment Whiteley Research Inc., dated 4/12/99 (updated for 2.3.3, mosis 7.2)
Comment Whiteley Research Inc., dated 5/9/99 (added scripts)
Comment Whiteley Research Inc., dated 8/6/99 (oct tweeks)
Comment Whiteley Research Inc., dated 2/13/00 (vias in CCG, layer order, \
  script tweeks)
Comment Whiteley Research Inc., dated 9/28/00 (Path definitions commented out)
Comment Whiteley Research Inc., dated 2/1/01 (Enhancements to scripts)
Comment Whiteley Research Inc., dated 2/14/01 (deleted plot5 print block)
Comment Whiteley Research Inc., dated 3/4/01 (Added Thickness, etc to layers, \
  these are guesses for testing extraction)
Comment Whiteley Research Inc., dated 7/7/01 (Added precision to Measure lines)
Comment Whiteley Research Inc., dated 8/6/01 (Interactive DRC limits changed)
Comment Whiteley Research Inc., dated 9/16/01 (driver blocks updated)
Comment Whiteley Research Inc., dated 3/10/02 "Merge M" -> "Merge"
```

```
Comment Whiteley Research Ibc., dated 4/12/02 Updated FN key defaults
Comment Whiteley Research Ibc., dated 11/18/02 INIT --> PARM
Comment Whiteley Research Ibc., dated 2/6/03 Set HardCopyDefResol

Technology scmos

!Set FlattenPrefix $$
Set lambda = .4
Define L(x) eval($(lambda)*x)
Define A(x) eval($(lambda)*$(lambda)*x*x)
#Define Electrode
#Define Analog
Define TripleMetal
#Define QuadMetal
#Define CapWell
#Define MEMS
Define SUBM
#Define HighVoltage
#Define TightMetal
#Define PolyCap
Define OldContacts

# The search paths can be defined here
#Path ( . )
#LibPath ( . /usr/local/lib/xictools/xic/startup )
#HelpPath ( /usr/local/lib/xictools/xic/help )
#ScriptPath ( /usr/local/lib/xictools/xic/scripts )

Script xstr
str = PrintString(L(2))
glen = AskReal("Gate length? ", str)
p1[2]
p2[2]
ShowPrompt("Click to anchor transistor")
if !Point(p1)
    Exit()
end
ShowPrompt("Click to finish")
ShowGhost(7)
if !Point(p2)
    Exit()
end
ShowGhost(0)
ovl = L(3)
ext = L(2)
cl = GetCurLayer();
if (abs(p1[0] - p2[0]) > abs(p1[1] - p2[1]))
    x1 = p1[0]
    x2 = p2[0]
    if (x1 > x2)
        x3 = x1 + ext
```

```
        x4 = x2 - ext
    else
        x3 = x1 - ext
        x4 = x2 + ext
    end
    y1 = p1[1]
    if (p2[1] > p1[1])
        y2 = y1 + glen
        y3 = y1 - ovl
        y4 = y1 + glen + ovl
    else
        y2 = y1 - glen
        y3 = y1 + ovl
        y4 = y1 - glen - ovl
    end
    SetCurLayer("CPG")
    Box(x3, y1, x4, y2)
    SetCurLayer("CAA")
    Box(x1, y3, x2, y4)
else
    y1 = p1[1]
    y2 = p2[1]
    if (y1 > y2)
        y3 = y1 + ext
        y4 = y2 - ext
    else
        y3 = y1 - ext
        y4 = y2 + ext
    end
    x1 = p1[0]
    if (p2[0] > p1[0])
        x2 = x1 + glen
        x3 = x1 - ovl
        x4 = x1 + glen + ovl
    else
        x2 = x1 - glen
        x3 = x1 + ovl
        x4 = x1 - glen - ovl
    end
    SetCurLayer("CPG")
    Box(x1, y3, x2, y4)
    SetCurLayer("CAA")
    Box(x3, y1, x4, y2)
end
#state = DRCstate(FALSE)
Commit()
#DRCstate(state)
SetCurLayer(cl)
EndScript

Script via_mfs
```

```
ShowPrompt("Click to place (lower left)")
PushGhostBox(-L(1), -L(1), L(3), L(3))
PushGhostBox(0, 0, L(2), L(2))
UseTransform(1, 0, 0)
cl = GetCurLayer();
while (1)
    ShowGhost(8)
    a[2]
    if !Point(a)
        ShowPrompt("")
        SetCurLayer(cl)
        Exit()
    end
    ShowGhost(0);
    UseTransform(1, a[0], a[1])
    SetCurLayer("CMF")
    Box(-L(1), -L(1), L(3), L(3))
    SetCurLayer("CVA")
    Box(0, 0, L(2), L(2))
    SetCurLayer("CMS")
    Box(-L(1), -L(1), L(3), L(3))
    SetCurLayer(cl)
    Commit()
end
EndScript

Script via_mfa
ShowPrompt("Click to place (lower left)")
PushGhostBox(-L(1.5), -L(1.5), L(3.5), L(3.5))
PushGhostBox(0, 0, L(2), L(2))
PushGhostBox(-L(1), -L(1), L(3), L(3))
UseTransform(1, 0, 0)
cl = GetCurLayer();
while (1)
    ShowGhost(8)
    a[2]
    if !Point(a)
        ShowPrompt("")
        SetCurLayer(cl)
        Exit()
    end
    ShowGhost(0);
    UseTransform(1, a[0], a[1])
    cl = GetCurLayer();
    SetCurLayer("CAA")
    Box(-L(1.5), -L(1.5), L(3.5), L(3.5))
    SetCurLayer("CCA")
    Box(0, 0, L(2), L(2))
    SetCurLayer("CMF")
    Box(-L(1), -L(1), L(3), L(3))
    SetCurLayer(cl)
```

```
    Commit()
end
EndScript

Script via_mfp
ShowPrompt("Click to place (lower left)")
PushGhostBox(-L(1.5), -L(1.5), L(3.5), L(3.5))
PushGhostBox(0, 0, L(2), L(2))
PushGhostBox(-L(1), -L(1), L(3), L(3))
UseTransform(1, 0, 0)
cl = GetCurLayer();
while (1)
    ShowGhost(8)
    a[2]
    if !Point(a)
        ShowPrompt("")
        SetCurLayer(cl)
        Exit()
    end
    ShowGhost(0);
    UseTransform(1, a[0], a[1])
    cl = GetCurLayer();
    SetCurLayer("CPG")
    Box(-L(1.5), -L(1.5), L(3.5), L(3.5))
    SetCurLayer("CCP")
    Box(0, 0, L(2), L(2))
    SetCurLayer("CMF")
    Box(-L(1), -L(1), L(3), L(3))
    SetCurLayer(cl)
    Commit()
end
EndScript

ElecLayer SCED
StreamOut 1 0
RGB 225 225 225
AltRGB 225 225 225
Filled y
AltFilled y

ElecLayer NAME
StreamOut 4 0
RGB 255 230 175
AltRGB 255 230 175
Filled n
AltFilled n

ElecLayer MODL
StreamOut 5 0
RGB 175 225 175
AltRGB 175 225 175
```

```
Filled n
AltFilled n

ElecLayer VALU
StreamOut 6 0
RGB 255 225 225
AltRGB 255 225 225
Filled n
AltFilled n

ElecLayer PARM
StreamOut 7 0
RGB 200 175 225
AltRGB 200 175 225
Filled n
AltFilled n

ElecLayer NODE
StreamOut 8 0
RGB 255 255 225
AltRGB 255 255 225
Filled n
AltFilled n

ElecLayer ETC1
StreamOut 2 0
RGB 125 225 225
AltRGB 125 225 225
Filled n
AltFilled n

ElecLayer ETC2
StreamOut 3 0
RGB 225 225 125
AltRGB 225 225 125
Filled n
AltFilled n

ElecLayer TERM
StreamOut 9 0
RGB 225 225 225
AltRGB 225 225 225
Filled n
Invisible
AltFilled n
AltInvisible

IfDef TightMetal
DrcTest tmMinSpace min thr lyr
EdgeMin %thr%
Test Outside %min% %lyr%
```

```
Test Inside %thr% !%lyr%
Evaluate fail = DRCuserTest(0) & !DRCuserTest(1)
End
Endif

IfDef HighVoltage
LayerName CVP
StreamIn 21, 0-255
StreamOut 21 0

LayerName CVN
StreamIn 22, 0-255
StreamOut 22 0
Endif

IfDef MEMS
LayerName COP
StreamIn 23, 0-255
StreamOut 23 0

LayerName CPS
StreamIn 24, 0-255
StreamOut 24 0
Endif

IfDef NOTUSED
LayerName XP
StreamIn 26, 0-255
StreamOut 26 0

LayerName CFI
StreamIn 27, 0-255
StreamOut 27 0
Endif

IfDef PolyCap
LayerName CPC
StreamIn 28, 0-255
StreamOut 28 0
MinWidth L(8) #23.1 CPC minimum width L(8)
MinSpace L(4) #23.2 CPC minimum space L(4)
MinSpaceTo CAA L(8) #23.3 CPC minimum space to CAA L(8)
MinSpaceFrom CPG L(3) #23.4 CPC minimum extension from CPG L(3)
MinNoOverlap CCG L(2) #23.5 CPC overlap around contact L(2)
MinNoOverlap Region CPG L(2) #23.6 CPG overlap around cap contact L(2)
MinSpaceTo Region CCG CPG L(2) #23.7 CPC contact to CPG L(2)
MinSpaceTo CMF L(4) #23.8 minimum space to CMF L(4)
MinSpaceTo CMS L(2) #23.9 minimum space to CMS L(2)
Endif

IfDef SilicideBlock
```

```
LayerName CSB
StreamIn 29, 0-255
StreamOut 29 0
MinWidth L(4) #20.1 CSB minimum width L(4)
MinSpace L(4) #20.2 CSB minimum space L(4)
NoOverlap CCG #20.3 No contacts on CSB
MinSpaceTo CCG L(2) #20.3 CBS minimum space to contact L(2)
MinSpaceTo CAA L(2) #20.4 CBS minimum space to CAA L(2)
MinSpaceTo CPG L(2) #20.5 CBS minimum space to CPG L(2)
MinWidth Region CPG L(5) #20.7 minimum CPG width over CSB L(5)
MinSpace Region CPG L(7) #20.8 minimum CPG space over CSB L(7)
MinNoOverlap CPG L(2) #20.9 CSB minimum extension from CPG L(2)
Endif

LayerName CWP
StreamIn 41, 0-255
StreamOut 41 0
RGB 46 132 22
AltRGB 46 132 22
Filled 00 7e 7e 66 66 7e 7e 00
AltFilled n
IfDef SUBM
MinWidth L(12) #1.1 CWP minimum width L(12) (subm option)
MinSpace L(18) #1.2 CWP minimum space L(18) (subm option)
Else
MinWidth L(10) #1.1 CWP minimum width L(10)
MinSpace L(9) #1.2 CWP minimum space L(9)
Endif

LayerName CWN
StreamIn 42, 0-255
StreamOut 42 0
RGB 137 124 120
AltRGB 137 124 120
Filled 00 7e 7e 66 66 7e 7e 00 outline
AltFilled n
IfDef SUBM
MinWidth L(12) #1.1 CWN minimum width L(12) (subm option)
MinSpace L(18) #1.2 CWN minimum space L(18) (subm option)
Else
MinWidth L(10) #1.1 CWN minimum width L(10)
MinSpace L(9) #1.2 CWN minimum space L(9)
Endif

LayerName CSP
StreamIn 44, 0-255
StreamOut 44 0
RGB 255 170 174
AltRGB 255 170 174
Filled 40 20 10 08 04 02 01 80 outline
AltFilled n
```

```
MinWidth L(2) #4.4 CSP minimum width L(2)
MinSpace L(2) #4.4 CSP minimum space L(2)
MinSpaceTo CPG L(3) #4.1 CSP minimum space to CPG L(3)
MinNoOverlap CAA L(2) #4.2 CSP minimum extension from CAA L(2)
IfDef OldContacts
MinNoOverlap CCA L(1) #4.3 CSP minimum extension from CCA L(1)
Else
MinNoOverlap CCG L(1) #4.3 CSP minimum extension from CCG L(1)
Endif

LayerName CSN
StreamIn 45, 0-255
StreamOut 45 0
RGB 61 211 186
AltRGB 61 211 186
Filled 40 20 10 08 04 02 01 80 outline
AltFilled n
MinWidth L(2) #4.4 CSN minimum width L(2)
MinSpace L(2) #4.4 CSN minimum space L(2)
MinSpaceTo CPG L(3) #4.1 CSN minimum space to CPG L(3)
MinNoOverlap CAA L(2) #4.2 CSN minimum extension from CAA L(2)
IfDef OldContacts
MinNoOverlap CCA L(1) #4.3 CSN minimum extension from CCA L(1)
Else
MinNoOverlap CCG L(1) #4.3 CSN minimum extension from CCG L(1)
Endif

LayerName CAA
StreamIn 43, 0-255
StreamOut 43 0
RGB 254 255 1
AltRGB 254 255 1
Filled 01 82 44 28 10 28 44 82 outline
AltFilled n
Conductor Exclude CPG
WireWidth L(3)
MinWidth L(3) #2.1 CAA minimum width L(3)
MinSpace L(3) #2.2 CAA minimum space L(3)
IfDef SUBM
MinSpaceFrom Region CSN CWP L(6) #2.3 S/D to CWP well edge L(6) (subm option)
MinSpaceFrom Region CSP CWN L(6) #2.3 S/D to CWN well edge L(6) (subm option)
Else
MinSpaceFrom Region CSN CWP L(5) #2.3 S/D to CWP well edge L(5)
MinSpaceFrom Region CSP CWN L(5) #2.3 S/D to CWN well edge L(5)
Endif
MinSpaceFrom Region CSP CWP L(3) #2.4 contact to CWP well edge L(3)
MinSpaceFrom Region CSN CWN L(3) #2.4 contact to CWN well edge L(3)
MinNoOverlap CPG L(3) #3.4 CAA minimum extension from CPG L(3)

LayerName CPG
StreamIn 46, 0-255
```

```
StreamOut 46 0
RGB 255 85 82
AltRGB 255 85 82
Filled 18 3c 7e e7 e7 7e 3c 18 outline
AltFilled n
Routing
Thickness 0.1
WireWidth L(2)
MinWidth L(2) #3.1 CPG minimum width L(2)
IfDef SUBM
MinSpace L(3) #3.2 CPG minimum space L(3) (subm option)
Else
MinSpace L(2) #3.2 CPG minimum space L(2)
Endif
MinNoOverlap CAA L(2) #3.3 CPG minimum extension from CAA L(2)
MinSpaceTo CAA L(1) #3.5 CPG minimum space to CAA L(1)

LayerName CMF
StreamIn 49, 0-255
StreamOut 49 0
RGB 0 174 253
AltRGB 0 174 253
Filled aa 55 aa 55 aa 55 aa 55 outline
AltFilled n
Routing
Thickness 0.4
WireWidth L(3)
MinWidth L(3) #7.1 CMF minimum width L(3)
MinSpace L(3) #7.2.a CMF minimum space L(3)
IfDef TightMetal
tmMinSpace L(2) L(3) CMF #7.2.b CMF minimum space L(2) (Tight Metal)
Endif
IfDef OldContacts
MinNoOverlap CCA|CCP L(1) #7.3 CMF minimum extension from CCA or CCP L(1)
Else
MinNoOverlap CCG L(1) #7.3 CMF minimum extension from contact L(1)
Endif

IfDef OldContacts
LayerName CCP
StreamIn 47, 0-255
StreamOut 47 0
RGB 233 151 218
AltRGB 233 151 218
Filled y
AltFilled n
Via CPG CMF
Thickness 0.2
EpsRel 3.8
MinWidth L(2)  #5.1 CCP exact size L(2) X L(2)
MaxArea A(2)   #5.1 CCP exact size L(2) X L(2)
```

```
MinSpaceFrom CPG L(1.5) #5.2 CCP minimum from CPG edge L(1.5)
IfDef SUBM
MinSpace L(3) #5.3 CCP minimum space L(3) (subm option)
Else
MinSpace L(2) #5.3 CCP minimum space L(2)
Endif
MinSpaceTo CAA L(2) #5.4 CCP minimum space to CAA L(2)

LayerName CCA
StreamIn 48, 0-255
StreamOut 48 0
RGB 253 253 65
AltRGB 253 253 65
Filled y
AltFilled y
Via CAA CMF
MinWidth L(2)  #6.1 CCA exact size L(2) X L(2)
MaxArea A(2)   #6.1 CCA exact size L(2) X L(2)
MinSpaceFrom CAA L(1.5) #6.2 CCA minimum from CAA edge L(1.5)
IfDef SUBM
MinSpace L(3) #6.3 CCA minimum space L(3) (subm option)
Else
MinSpace L(2) #6.3 CCA minimum space L(2)
Endif
MinSpaceTo CPG L(2) #6.4 CCA minimum space to CPG L(2)

Else

LayerName CCG
StreamIn 25, 0-255
StreamOut 25 0
RGB 253 253 65
AltRGB 253 253 65
Filled y
AltFilled y
Via CPG CMF
Via CAA CMF
MinWidth L(2)  #5.1, 6.1 Contact exact size L(2) X L(2)
MaxArea A(2)   #5.1, 6.1 Contact exact size L(2) X L(2)
MinSpaceFrom CPG L(1.5) #5.2 Contact minimum from CPG edge L(1.5)
MinSpaceFrom CAA L(1.5) #6.2 Contact minimum from CAA edge L(1.5)
IfDef SUBM
MinSpace L(3) #5.3, 6.3 Contact minimum space L(3) (subm option)
Else
MinSpace L(2) #5.3, 6.3 Contact minimum space L(2)
Endif
MinSpaceTo CAA L(2) #5.4 Contact minimum space to CAA L(2)
MinSpaceTo CPG L(2) #6.4 Contact minimum space to CPG L(2)
Endif

LayerName CMS
```

```
StreamIn 51, 0-255
StreamOut 51 0
RGB 205 0 205
AltRGB 205 0 205
Filled 2a 15 8a 45 a2 51 a8 54 outline
AltFilled n
Routing
Thickness 0.6
WireWidth L(3)
MinWidth L(3) #9.1 CMS minimum width L(3)
MinSpace L(4) #9.2.a CMS minimum space L(4)
IfDef TightMetal
tmMinSpace L(3) L(4) CMS #9.2.b CMS minimum space L(3) (Tight Metal)
Endif
MinNoOverlap CVA L(1) #9.3 CMS minimum extension from CVA L(1)

LayerName CVA
StreamIn 50, 0-255
StreamOut 50 0
RGB 240 218 181
AltRGB 240 218 181
Filled y
AltFilled n
Via CMF CMS
Thickness 0.6
EpsRel 3.8
MinWidth L(2) #8.1 CVA exact size L(2) X L(2)
MaxArea A(2)  #8.1 CVA exact size L(2) X L(2)
MinSpace L(3) #8.2 CVA minimum space L(3)
MinSpaceFrom CMF L(1) #8.3 CVA minimum from edge of CMF L(1)
IfDef OldContacts
MinSpaceTo CCA|CCP L(2) #8.4 CVA minimum space to CCA or CCP L(2)
Else
MinSpaceTo CCG L(2) #8.4 CVA minimum space to contact L(2)
Endif
MinSpaceTo CAA|CPG L(2) #8.5 CVA minimum space to CAA or CPG L(2)

LayerName COG
StreamIn 52, 0-255
StreamOut 52 0
RGB 255 250 107
AltRGB 255 250 107
Filled n
AltFilled n
Thickness 1.0
EpsRel 3.2
MinWidth 75 #10.1, 10.2 COG minimum pad width
MinSpaceFrom CMF|CMS 6.0 #10.3 metal extension minimum from COG

IfDef Electrode
LayerName CEL
```

```
StreamIn 56, 0-255
StreamOut 56 0
MinWidth L(3) #11.1 CEL minimum width L(3)
MinSpace l(3) #11.2 CEL minimum space L(3)
MinSpaceFrom CPG L(2) #11.3 CEL minimum from CPG edge L(2)
MinSpaceTo CAA L(2) #11.4 CEL minimum space to CAA L(2)
IfDef OldContacts
MinSpaceTo CCP L(3) #11.5 CEL Minimum space to CCP L(3)
Else
MinSpaceTo CCG L(3) #11.5 CEL Minimum space to CCG L(3)
Endif
Endif

IfDef Analog
LayerName CEL
StreamIn 56, 0-255
StreamOut 56 0
MinWidth L(3) #11.1 CEL minimum width L(3)
MinSpace l(3) #11.2 CEL minimum space L(3)
MinSpaceFrom CPG L(2) #11.3 CEL minimum from CPG edge L(2)
MinSpaceTo CAA L(2) #11.4 CEL minimum space to CAA L(2)
IfDef OldContacts
MinSpaceTo CCP L(3) #11.5 CEL Minimum space to CCP L(3)
Else
MinSpaceTo CCG L(3) #11.5 CEL Minimum space to CCG L(3)
Endif

LayerName CCD
StreamIn 57, 0-255
StreamOut 57 0

LayerName CBA
StreamIn 58, 0-255
StreamOut 58 0
MinWidth L(3) #18.1 CBA minimum width L(3)
#MinSpaceFrom CAA L(0.5) #18.2 CBA minimum from CAA edge L(0.5)
Endif

IfDef CapWell
LayerName CWC
StreamIn 58, 0-255
StreamOut 58 0
IfDef SUBM
MinWidth L(12) #17.1 CWC minimum width L(12) (subm option)
MinSpace L(18) #17.2 CWC minimum space L(18) (subm option)
MinSpaceTo CAA L(6) #17.3 CWC minimum space to CAA L(6) (subm option)
MinSpaceFrom CAA L(6) #17.4 CWC minimum from CAA edge L(6) (subm option)
Else
MinWidth L(10) #17.1 CWC minimum width L(10)
MinSpace L(9) #17.2 CWC minimum space L(9)
MinSpaceTo CAA L(5) #17.3 CWC minimum space to CAA L(5)
```

```
MinSpaceFrom CAA L(5) #17.4 CWC minimum from CAA edge L(5)
Endif
Endif

IfDef TripleMetal
LayerName CVS
StreamIn 59, 0-255
StreamOut 59 0
RGB 185 156 159
AltRGB 185 156 159
Filled y
AltFilled n
Via CMT CMS
MinWidth L(2) #14.1 CVS exact size L(2) X L(2)
MaxArea A(2)  #14.1 CVS exact size L(2) X L(2)
MinSpace L(3) #14.2 CVS minimum space L(3)
MinSpaceFrom CMS L(1) #14.3 CVS minimum to CMS edge L(1)
MinSpaceTo CVA L(2) #14.4 CVS minimum space to CVA L(2)

LayerName CMT
StreamIn 61, 0-255
StreamOut 61 0
RGB 231 247 236
AltRGB 172 167 148
Filled 88 44 02 11 88 40 22 11 outline
AltFilled n
Conductor
IfDef SUBM
MinWidth L(5) #15.1 CMT minimum width L(5) (subm option)
MinSpace L(3) #15.2 CMT minimum space L(3) (subm option)
Else
MinWidth L(6) #15.1 CMT minimum width L(6)
MinSpace L(4) #15.2 CMT minimum space L(4)
Endif
MinNoOverlap CVS L(2) #15.3 minimum extension from CVS L(2)
Endif

IfDef QuadM
LayerName CVS
StreamIn 59, 0-255
StreamOut 59 0
Via CMT CMS
MinWidth L(2) #14.1 CVS exact size L(2) X L(2)
MaxArea A(2)  #14.1 CVS exact size L(2) X L(2)
MinSpace L(3) #14.2 CVS minimum space L(3)
MinSpaceFrom CMS L(1) #14.3 CVS minimum to CMS edge L(1)
MinSpaceTo CVA L(2) #14.4 CVS minimum space to CVA L(2)

LayerName CMT
StreamIn 61, 0-255
StreamOut 61 0
```

```
Conductor
MinWidth L(3) #15.1 CMT minimum width L(3)
MinSpace L(4) #15.2.a CMT minimum space L(4)
tmMinSpace L(3) L(4) CMT #15.2.b CMS minimum space L(3)
MinNoOverlap CVS L(1) #15.3 minimum extension from CVS L(1)

LayerName CVT
StreamIn 30, 0-255
StreamOut 30 0
Via CMQ CMT
MinWidth L(2) #21.1 CVT exact size L(2) X L(2)
MaxArea A(2)  #21.1 CVT exact size L(2) X L(2)
MinSpace L(4) #21.2 CVT minimum space L(4)
MinSpaceFrom CMT L(1) #21.3 CVT minimum to CMT edge L(1)

LayerName CMQ
StreamIn 31, 0-255
StreamOut 31 0
Conductor
MinWidth L(6) #22.1 CMQ minimum width L(6)
MinSpace L(6) #22.2 CMQ minimum space L(6)
MinNoOverlap CVT L(4) #22.3 CMQ minimum extension from CVT L(4)
Endif

LayerName NDRC
StreamIn 21, 0-255
StreamOut 21 0
RGB 175 175 175
AltRGB 175 175 175
Filled n
AltFilled n

Device
Name nmos
Prefix Mn_
Body (CAA&CPG)&CSN
Contact s CAA CAA&(!CPG)
Contact d CAA CAA&(!CPG)
Contact g CPG CAA&CPG
Permute d s
Bloat 0.1
Merge
Measure Length CBWidth*1e-6 4
Measure Width CBNWidth*1e-6 4
Measure AS CArea.0.CAA * 1e-12 4
Measure AD CArea.1.CAA * 1e-12 4
Measure PS CPerim.0.CAA * 1e-6 4
Measure PD CPerim.1.CAA * 1e-6 4
LVS Length l
LVS Width w
LVS AS as
```

```
LVS AD ad
LVS PS ps
LVS PD pd
Model nmos
Initc l=%me%Length w=%me%Width ad=%me%AD pd=%me%PD as=%me%AS ps=%me%PS
Spice %n% %c%d %c%g %c%s NSUB %model% %initc%
Cmput NMOS device %n%, Length: %me%Length  Width: %me%Width  AS: %me%AS \
PS: %me%PS  AD: %me%AD  PD: %me%PD
End

Device
Name pmos
Prefix Mp_
Body (CAA&CPG)&CSP
Contact s CAA CAA&(!CPG)
Contact d CAA CAA&(!CPG)
Contact g CPG CAA&CPG
Permute d s
Bloat 0.1
Merge
Measure Length CBWidth * 1e-6 4
Measure Width CBNWidth * 1e-6 4
Measure AS CArea.0.CAA * 1e-12 4
Measure AD CArea.1.CAA * 1e-12 4
Measure PS CPerim.0.CAA * 1e-6 4
Measure PD CPerim.1.CAA * 1e-6 4
LVS Length l
LVS Width w
LVS AS as
LVS AD ad
LVS PS ps
LVS PD pd
Model pmos
Initc l=%me%Length w=%me%Width ad=%me%AD pd=%me%PD as=%me%AS ps=%me%PS
Spice %n% %c%d %c%g %c%s PSUB %model% %initc%
Cmput PMOS device %n%, Length: %me%Length  Width: %me%Width  AS: %me%AS \
PS: %me%PS  AD: %me%AD  PD: %me%PD
End

DRC y
DrcMaxErrors 0
DrcInterMaxObjs 1000
DrcInterMaxTime 5000
DrcInterSkipInst n
MergeOnRead n
InToLower n
OutToUpper n
GridSpacing 1
ShowGrid y
GridOnBottom y
PhysGridStyle 204
```

```
ElecGridStyle 0
PhysSnapping 2
ElecSnapping 1
Constrain45 n
Expand 0
DisplayAllText y
ShowPhysProps n
LabelAllInstances 1
ShowInstanceMark n
ShowContext y
ShowTinyBB y
RoundFlashSides 40
BeepVolume 0
BoxLineStyle 3640
Background 0 0 0
Highlighting white
InstanceBox turquoise
InstanceName pink
InstanceSize salmon
PhysCoarseGrid 0 215 215
ElecCoarseGrid 255 208 136
PhysFineGrid 0 0 207
ElecFineGrid 0 245 249
AltPhysCoarseGrid 0 215 215
AltElecCoarseGrid 255 208 136
AltPhysFineGrid 0 0 207
AltElecFineGrid 0 245 249
AltShowGrid n
AltGridStyle 204
AltDriver postscript_bitmap
AltBackground 0 0 0
F1key box
F2key wire
F3key polyg
F4key label
F5key open
F6key cells
F7key prpty
F8key desel

HardCopyDevice postscript_line_draw
HardCopyResol 72 75 100 150 200 300 400
HardCopyDefResol 3
HardCopyPixWidth 0
HardCopyDefWidth 7.8
HardCopyDefHeight 10.3
HardCopyDefXoff 0.25
HardCopyDefyoff 0.25
HardCopyMinWidth 1
HardCopyMinHeight 1
HardCopyMinXoff 0
```

```
HardCopyMinyoff 0
HardCopyMaxWidth 23.5
HardCopyMaxHeight 23.5
HardCopyMaxXoff 22
HardCopyMaxyoff 22

HardCopyDevice postscript_line_draw_color
HardCopyResol 72 75 100 150 200 300 400
HardCopyDefResol 3
HardCopyPixWidth 0
HardCopyDefWidth 7.8
HardCopyDefHeight 10.3
HardCopyDefXoff 0.25
HardCopyDefyoff 0.25
HardCopyMinWidth 1
HardCopyMinHeight 1
HardCopyMinXoff 0
HardCopyMinyoff 0
HardCopyMaxWidth 23.5
HardCopyMaxHeight 23.5
HardCopyMaxXoff 22
HardCopyMaxyoff 22

HardCopyDevice postscript_bitmap
HardCopyResol 72 75 100 150 200 300 400
HardCopyDefResol 3
HardCopyPixWidth 0
HardCopyDefWidth 7.9
HardCopyDefHeight 10.3
HardCopyDefXoff 0.25
HardCopyDefyoff 0.25
HardCopyMinWidth 1
HardCopyMinHeight 1
HardCopyMinXoff 0
HardCopyMinyoff 0
HardCopyMaxWidth 23.5
HardCopyMaxHeight 23.5
HardCopyMaxXoff 22
HardCopyMaxyoff 22

HardCopyDevice postscript_bitmap_encoded
HardCopyResol 72 75 100 150 200 300 400
HardCopyDefResol 3
HardCopyPixWidth 0
HardCopyDefWidth 7.9
HardCopyDefHeight 10.3
HardCopyDefXoff 0.25
HardCopyDefyoff 0.25
HardCopyMinWidth 1
HardCopyMinHeight 1
HardCopyMinXoff 0
```

```
HardCopyMinyoff 0
HardCopyMaxWidth 23.5
HardCopyMaxHeight 23.5
HardCopyMaxXoff 22
HardCopyMaxyoff 22

HardCopyDevice postscript_bitmap_color
HardCopyResol 72 75 100 150 200 300 400
HardCopyDefResol 3
HardCopyPixWidth 0
HardCopyDefWidth 7.9
HardCopyDefHeight 10.3
HardCopyDefXoff 0.25
HardCopyDefyoff 0.25
HardCopyMinWidth 1
HardCopyMinHeight 1
HardCopyMinXoff 0
HardCopyMinyoff 0
HardCopyMaxWidth 23.5
HardCopyMaxHeight 23.5
HardCopyMaxXoff 22
HardCopyMaxyoff 22

HardCopyDevice postscript_bitmap_color_encoded
HardCopyResol 72 75 100 150 200 300 400
HardCopyDefResol 3
HardCopyPixWidth 0
HardCopyDefWidth 7.9
HardCopyDefHeight 10.3
HardCopyDefXoff 0.25
HardCopyDefyoff 0.25
HardCopyMinWidth 1
HardCopyMinHeight 1
HardCopyMinXoff 0
HardCopyMinyoff 0
HardCopyMaxWidth 23.5
HardCopyMaxHeight 23.5
HardCopyMaxXoff 22
HardCopyMaxyoff 22

HardCopyDevice hpgl_line_draw_color
HardCopyPixWidth 0
HardCopyDefWidth 7.8
HardCopyDefHeight 10.3
HardCopyDefXoff 0.25
HardCopyDefyoff 0.25
HardCopyMinWidth 1
HardCopyMinHeight 1
HardCopyMinXoff 0
HardCopyMinyoff 0
HardCopyMaxWidth 23.5
```

```
HardCopyMaxHeight 23.5
HardCopyMaxXoff 22
HardCopyMaxyoff 22

HardCopyDevice hp_laser_pcl
HardCopyResol 75 100 150 300
HardCopyDefResol 2
HardCopyPixWidth 0
HardCopyDefWidth 7.8
HardCopyDefHeight 10.3
HardCopyDefXoff 0.25
HardCopyDefyoff 0.25
HardCopyMinWidth 1
HardCopyMinHeight 1
HardCopyMinXoff 0
HardCopyMinyoff 0
HardCopyMaxWidth 16.5
HardCopyMaxHeight 16.5
HardCopyMaxXoff 15
HardCopyMaxyoff 15

HardCopyDevice xfig_line_draw_color
HardCopyPixWidth 0
HardCopyDefWidth 4
HardCopyDefHeight 4
HardCopyDefXoff 0
HardCopyDefyoff 0
HardCopyMinWidth 1
HardCopyMinHeight 1
HardCopyMinXoff 0
HardCopyMinyoff 0
HardCopyMaxWidth 23.5
HardCopyMaxHeight 23.5
HardCopyMaxXoff 22
HardCopyMaxyoff 22

HardCopyDevice versatec_monochrome
HardCopyPixWidth 7040
HardCopyDefWidth 35
HardCopyDefHeight 0
HardCopyDefXoff 0
HardCopyDefyoff 0
HardCopyMinWidth 2
HardCopyMinHeight 2
HardCopyMinXoff 0
HardCopyMinyoff 0
HardCopyMaxWidth 35
HardCopyMaxHeight 240
HardCopyMaxXoff 30
HardCopyMaxyoff 0
```

```
HardCopyDevice versatec_color
HardCopyPixWidth 6848
HardCopyDefWidth 34
HardCopyDefHeight 0
HardCopyDefXoff 0
HardCopyDefyoff 0
HardCopyMinWidth 2
HardCopyMinHeight 2
HardCopyMinXoff 0
HardCopyMinyoff 0
HardCopyMaxWidth 34
HardCopyMaxHeight 240
HardCopyMaxXoff 30
HardCopyMaxyoff 0

HardCopyDevice wpm
HardCopyPixWidth 0
HardCopyDefWidth 4
HardCopyDefHeight 4
HardCopyDefXoff 0.25
HardCopyDefyoff 0.25
HardCopyMinWidth 1
HardCopyMinHeight 1
HardCopyMinXoff 0
HardCopyMinyoff 0
HardCopyMaxWidth 8
HardCopyMaxHeight 10.5
HardCopyMaxXoff 10
HardCopyMaxyoff 10

HardCopyDevice wp
HardCopyPixWidth 0
HardCopyDefWidth 4
HardCopyDefHeight 4
HardCopyDefXoff 0
HardCopyDefyoff 0
HardCopyMinWidth 1
HardCopyMinHeight 1
HardCopyMinXoff 0
HardCopyMinyoff 0
HardCopyMaxWidth 8
HardCopyMaxHeight 10.5
HardCopyMaxXoff 10
HardCopyMaxyoff 10

HardCopyDevice imlib
NoAlt
HardCopyPixWidth 0
HardCopyDefWidth 4
HardCopyDefHeight 4
HardCopyDefXoff 0
```

```
HardCopyDefyoff 0
HardCopyMinWidth 1
HardCopyMinHeight 1
HardCopyMinXoff 0
HardCopyMinyoff 0
HardCopyMaxWidth 16.5
HardCopyMaxHeight 16.5
HardCopyMaxXoff 15
HardCopyMaxyoff 15
```

A.2 Resource File

Under Unix/Linux, attribute colors can be set in an X resource file. This is a file named "`Xic`" (note the first letter is capitalized) located in the user's home directory. 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.1.7. 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 found in the X `rgb.txt` file (this works for Windows, too). These 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`".
- (Unix/Linux only) 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.

A.3 Symbol 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 **Conversion** – **Export** panel should be used to strip out all extensions, leaving only the physical layout.

The GDSII (Stream) format is owned by Cadence, Inc. It is described in the "Design Data Translators Reference," which is updated periodically. The CIF format is public domain.

In *Xic*, symbol names are not limited in length in the present release. Previous *Xic* releases limited symbol names to 32 characters, to match the GDSII release 3 specification. The name of an *Xic* symbol is always that of the native-format file from which it is read. The symbol name can contain any characters valid in a file name with the exception of the semicolon (';'), which is reserved in the CIF-like syntax. For portability, it is recommended that symbol names use only the GDSII allowed characters, which are the alpha-numerics plus '_', '$', and '?'.

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 those formats are 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.

A.3.1 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 **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 **Conversion − Export** 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 symbol 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.

Physical Mode Cell Properties

Certain features, such as template 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 **Convert** 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.

A.3.2 The CIF File Format

The format used in native symbol 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.

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.

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. This name is alphanumeric and is four characters in length 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.

A.3.3 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.

1. 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.

2. 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. If the boolean variable `CifOldWire` is set with the **!set** command or otherwise, generated output will use the old wire format, providing backwards compatibility. The old format used a separate property to define the wire style, which is less efficient than the new format. For example:

   ```
   5 7033 PATHTYPE 0;
   W 1000 10000 12000 20000 12000;
   ```

3. Labels are specified with a unique syntax:
   `94 <<`*label string*`>>` *x y code width height*`;`
   This is a further extension of a widely-used extension for labels, which does not have the *code*, *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 *code* entry specifies transformations applied to the label at the reference point, and other rendering information, as shown in the table.

bits 0-1	rotate the text about x,y 00 no rotation 01 90 degrees 10 180 degrees 11 270 degrees
bits 2-3	bit 2 = 1, mirror y after rotation bit 3 = 1, mirror x after mirror y
bit 4	bit 4 = 1, shift rotations to 45, 135, 225, 315 degrees
bits 5-6	horizontal justification, 00, 11 left, 01 center, 10 right
bits 7-8	vertical justification, 00, 11 bottom, 01 center, 10 top
bits 9-10	GDSII font number

4. Cell instance lines can be preceded by an array extension line of the form
   `1 Array` *nx 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.

5. Cell instances can be preceded by a magnification extension of the form
   `1 Magnify` *scale_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 should appear in physical cell descriptions only. It is unique to *Xic*.

6. Cell instances can also 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*.

7. 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 character of the physical cell. In *Xic* the cell terminator is the single character `E`. 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 character. Again, the terminating character is `E`, however if **Strip For Export** is active the terminator is `End` for compatibility reasons.

Xic normally uses a resolution of 1000 units per micron in unstripped CIF and native files, signaled with the addition of a comment line ahead of the first symbol definition:

```
(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 1000 units when writing a CIF file without the **Strip For Export** button active, and 100 units otherwise.

A.3.4 Native Symbol File Format

Native symbols use an extension to the CIF format, with one symbol per file. Each file can contain two cell definitions: one for physical geometry, and the second for electrical (schematic) data. The physical information is found first. If there is no electrical information, the electrical cell definition is not written. The parser will also recognize schematic files from the Jspice3 program, which have a similar format, but no physical cell definition. *Xic* will write an empty physical cell definition if there is no physical information (electrical only).

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 CIF comment giving the symbol name. The second line is a comment providing an ID field to be used with the RCS code control program. The RCS program is used to manage large projects with multiple designers. An optional comment line containing the creating program version and creation date follows. 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 electrical part of the cell 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*.

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.

A.3.5 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. CGX files can be used in skeleton mode, as with GDSII.

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.

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	

As in GDSII, string data are terminated with a null byte, with an additional null byte added if necessary to make the total length even.

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, 11 left, 01 center, 10 right
bits 6-7	vertical justification, 00, 11 bottom, 01 center, 10 top

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

The `width` gives the physical equivalent width of the text. The height is determined by the font used for rendering.

`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.

A.4 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
Property number string
...
Reference name path [cellname]
...
(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.

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.

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.

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.

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.

A.4.1 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
```

A.5 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.

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 in the device library file. First are the terminals, including the gnd device and the vcc device. The second class of device is the sources, including voltage and current sources, and dependent sources. These have no physical counterparts. The third class consists of the devices with physical counterparts, such as resistors and semiconductor devices. The fourth class is the subcircuit macros, which actually introduce a subcircuit into the SPICE output.

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.

The device library file handles "global properties". These properties appear at the beginning of the file, after the initial line but before the symbol definitions. The syntax is

```
Property number string
```

where Property appears literally, *number* is an integer, and the rest of the line constitutes the *string*. There can be any number of these lines.

Presently, there are two global properties recognized. The first is number 20, the LibSave property. 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 20 I c
```

This will produce lines in the SPICE output like

```
.save @isource[c]
```

for a current source named isource.

The second property recognized is 21, the GlobalNode property. This 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 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 "vcc" terminal, and modifying the terminal name to the node name (NSUB).

The rest of the file consists of multiple concatenated device specifications. There is no physical representation, and the resolution as shown (and as in the supplied `device.lib` file) is 100 units, unlike symbol files which use 1000 units per "micron". However, the `(RESOLUTION 1000);` comment can appear, which indicates 1000 units per "micron", as in ordinary cells. Each device has the following format:

(Symbol *symname*);
5 *property*;
5 ... ;
9 *symname*;
DS 0 1 1;
L SCED;
geometry ...
more layers/geometry ...
DF;
E

The format is extended CIF, as used in the electrical description of symbol files. 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 A.5.1), 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 B) 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 **Edit** 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 vcc device is a special non-physical object used for tying different parts of the circuit together without a wire, and for assigning node names. One can rename the vcc device to other names in the device library file. The default terminal name is the name given to the device. It is important that the name property begin with the character '@'. A call to the vcc device is hard-coded in the **Source SPICE** command (among others) device placement code, so the supplied vcc device should not be removed from the device library file. The name label of the terminal can be modified. Nodes with an attached terminal with the same name label are tied together in the SPICE output.

There are two special layers added to the layer table: NODE, and TERM. The label associated with vcc type terminals use the NODE layer. The TERM layer is normally invisible, and is intended to be used for a vcc terminal device to be used for node naming, but is invisible after placement (except for the label).

A.5.1 Editing Devices

Devices in the device library can be edited, while in electrical mode, by simply giving the device name to the **Edit** command or equivalent. When saving, with either **Save** or **Save As**, the **Library Device Parameters** pop-up will appear. This allows various defaults and parameters to be specified for the device, and allows it to be saved in a device library file or in a native cell file.

This panel will appear when a library device is saved while in either electrical or physical mode.

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`"). In this case, the current cell is internally marked as a library device, and the panel appears. This is one way to add new devices to the library, the other way is to edit an existing device and change the name. Of course, the cell should contain geometry appropriate for the device library, i.e., no physical data, no subcells, etc.

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.

The remainder of this section describes the controls found in the **Library Device Parameters** panel.

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 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 vcc terminal, 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.

If the current cell is not a SPICE device but rather a macro, which will instantiate a subcircuit (such as the "`opamp`" example in the supplied `device.lib` file) the SPICE Prefix *must* start with '`x`' or '`X`'.

A subcircuit added in this manner is expected to reference a `.subckt` macro in the model library. The name of the macro (not the file name) is given to instances of the device as a model property. A default model property can be supplied to the device. In the example, 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.

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 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.

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 B.4.6 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 is 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 Native Cell** button can be used to save the device as a symbol file in the current directory. When this is done, a message appears warning that it is not a good idea to have such files lying around,

as they can potentially conflict with the device library. The file should be moved somewhere safe, i.e., out of the search path. They can be copied into the device library file with a text editor to manually update the device library file.

A.6 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. 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 **Edit** 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 **Edit** command and saved, all offset tables are updated.

A.7 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:

in *Xic*	Xic
in *XicII*	Xic, XicII
in *WRspice*	WRspice
in all, under Windows	Windows

If *word* is defined, the text up to the next !!ELSE or !!ENDIF will be included in the topic, and any text following an !!ELSE up to !!ENDIF is discarded. If *word* is not 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 `!!ENDIF`. The `!!ELSE` is optional. The `!!SEEALSO` and `!!SUBTOPICS` lines can appear within the blocks.

Example:

```
!!HTML
   Here is some text.
!!IFDEF Xic
   You are reading this in Xic.
!!ELSE
!!IFDEF WRspice
   You are reading this in WRspice.
!!ELSE
   You are not reading this in Xic or WRspice.
!!ENDIF
!!ENDIF
```

`!!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`.

`!!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.

`!!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.

`!!SEEALSO` *keyword-list*
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, it is silently ignored. The keywords listed *must* be given in a `!!KEYWORD` lines, and not contain named anchor references.

`!!SUBTOPICS` *keyword-list*
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`.

A.7.1 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 `<a href="somekeyword#refname">blather</a>` 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: '`$`' expansion. If the first character of an anchor URL is '`$`', it is processed specially. The leading word is replaced with other text, either an internal path, or an environment variable. The internally recognized tokens are:

`$HELP`	replaced by first component of the help path
`$EXAMPLES`	replaced by path to examples
`$DOCS`	replaced by path to docs
`$SCRIPTS`	replaced by path to scripts

Otherwise, if it matches a variable in the environment, it is replaced by the environment variable value. If no match, it is left alone.

The paths are derived from the first component of the Help path. If the `HelpPath` variable is "`/usr/local/lib/xictools/xic/help /blather/help`", then the substitution paths are `/usr/local/lib/xictools/xic/examples`", etc.

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 "`/users/yourhome/mydir`", and "`~joe/joesdir`" might expand to "`/users/joe/joesdir`", etc.

In *Xic*, one can open *Xic* 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`

Native *Xic* files are only recognized if they have a `.xic` extension. In the case where an *Xic* file has no extension, or another extension, it will have to be copied or linked to provide the `.xic` extension before it can be accessed through anchor text. The anchor text is given in the normal way, e.g.

```
Click <a href="http://somewhere/lib/cell.gds">here</a> to view the design.
Click <a href="/usr/joe/library/joecell.gds">here</a> to view Joe's cell.
```

In addition, if a reference 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.

For example, the HTML text in a file to be viewed with the help system might contain the line

```
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*, a similar capability exists. 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, anchor references to files with a `.raw` extension will be loaded into *WRspice* as a *rawfile*, i.e., a plot data file, when the anchor is clicked.

This page intentionally left blank.

Appendix B

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 below. 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 6.1.2). These values should not be used for assigned properties.

B.1 Global Properties

The "Global" properties are added to the top level cell of a hierarchy. They save the grid parameters, plot points, and other information in the file, to use as defaults when the file is loaded for editing.

Grid Property: Property Number 7100
This property us used in physical mode only. The property string has the format

`grid` *resol snap*

where *resol* is the number of internal units per grid line, and *snap* is the number of snap points per grid line if positive, or grid lines per snap point if negative.

Run Property: Property Number 7101
The Run property is used in electrical mode only. The string specifies the default analysis command entered when the **run** button is pressed.

Plot Property: 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: 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 B.6.

B.2 Special Properties

When converting from GDSII, there are attributes used in GDSII that are not used by *Xic*. These attributes are saved as properties, and will be added to any subsequent GDSII file produced. Also, the wire end style is saved as a property of wires, since the CIF format does not provide a variable for this.

LabelSize Property: Property number 7180
: This property is added to labels when writing to GDSII, and saves the label width and height. The string has the format

    `width` *width* `height` *height*

  where *width* and *height* are in internal units.

Text Property: 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: Property number 7033
: This property is used for physical mode wires of nonzero width which have a nondefault 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 not added to wire objects when reading GDSII, but is added to wires in native and CIF output if the CifOldWire variable is set. The wire end style is now included in the wire specification (see A.3.3) in the present default syntax.

B.3 Physical Mode Properties

There are a few properties that may be applied to physical cells or objects in support of certain *Xic* features.

Flags Property: 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	
4	USER2	
5	USER3	
6	USER4	
7	USER5	

This property can be set in physical mode with the **Cell Properties** editor.

When the ExtractOpaque variable is set, the OPAQUE flag is ignored.

The remaining properties support template cells.

Template Name Property: Property number 7197
This property is assigned by *Xic* to cells and instances derived from template cells. It provides the name of the template cell from which the current cell was derived.

Template Params Property: Property number 7198
This property is assigned by the user to template cells. It contains a list of parameter assignments to provide default values to the template cell. Instances and masters obtained from the template will also contain this property, assigned by *Xic*, containing the actual parameter values used for instantiation.

Template Script Property: Property number 7199
This property is assigned by the user to template cells. It contains the script which is executed when the template cell is instantiated.

B.4 Device Properties Set By *Xic*

The properties that are used by *Xic* in electrical mode are described below. There are basically two classes of properties: those that are set by *Xic*, to be described in this section, and those that are set by the user while running *Xic*. Both types may be required when editing the device library file. *Xic* may add fields to the properties provided in the device library file, and/or fill in "place holder" fields, in the property specifications found in symbol files.

The electrical property values use the integers 1-20. The values 6–9, and 20 are not currently used, but are reserved for future use.

Below, property fields that are shown in square brackets "[...]" are optional. Those shown in curly brackets "{...}" are added to specification lines in symbol files, and should *not* appear in device library files. All of these properties are restricted to electrical cell definitions.

The following properties are normally set by *Xic*.

B.4.1 Node Property: Property number 10

The node property defines a connection point of the device. It appears in device cell definitions and instance references.

5 10 *circuit_node device_node x y* [*name phys_x phys_y*] {*flags lname refname*};

The *circuit_node* is always set to -1 in the device library file. The *device_node* is a count, starting at zero, of the nodes assigned to the device or subcircuit. The node properties for a cell definition and its instances must share the same *device_node*'s, i.e., there must be a one to one correspondence between listings (*Xic* maintains this). Device node zero is the reference node. When the device is placed in a drawing, the location of the reference node corresponds to where the user clicks. The device node numbers should be sequential and unique, and as mentioned start with zero. The x and y parameters specify the location of the terminal in database coordinates relative to the device cell origin. The default grid spacing is 1 micron (this is a vestige of the physical mode of *Xic*), thus it is advisable to place nodes at coordinates divisible by the resolution, which is 100 unless specified otherwise. The typical device in the library has a maximum dimension corresponding to 10 microns. The origin of the cell is arbitrary, the node locations should match the geometry of the cell.

If the node property is applied to a cell definition rather than an instance, and the device or subcircuit has a symbolic representation, the x and y parameters are of the form

x:syx y:syy

i.e., two fields of two numbers separated by colons. The first numbers of the two fields (x, y) are the coordinates of the terminal in the schematic representation. The second two numbers (syx, syy) are the coordinates of the terminal in the symbolic representation.

The bracketed quantities are optional, and are used by the netlist extraction subsystem. The *name* is a short identifying name given to the node, which should be unique among the nodes of a device or subcircuit. The next two parameters are the coordinates in the physical cell where the node is referenced, and should always be set to 0 in the device library file. If these three parameters are given, the device terminal is expected to have an actual physical location, meaning that the device is a "real" device, such as a resistor, rather than a simulation model such as a voltage source.

It is not necessarily true that all nodes of the device either have or don't have the optional parameters given in a device library file. The phase node of a Josephson junction, for example, does not have the parameters given, since this node has no physical counterpart. The other two nodes do have the optional parameters given, since these are the physical connection points. A side-effect is that in SPICE files created 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 (described below) should also be given in that case.

If the bracketed parameters were given in the device library file, then instances of the device in symbol files may contain the last three fields. The *flags* field has the following bits possibly set:

bit 1 set if the terminal is not movable (unused)
bit 2 set if the terminal is virtual

The *lname* is the name of the layer to which the physical terminal is attached. It is the layer name from the technology file of a physical layer which has the Conductor attribute.

The *refname* is an optional keyword that specifies a terminal type. Valid keywords are:

```
input (default)
output
inout
tristate
clock
outclock
supply
outsupply
ground
```

B.4.2 Name Property: 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 **Properties** command or equivalent, that name will be used in SPICE output. Otherwise, the name prefix from the device library file 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).

5 11 *prefix*{.*assigned_name*} *devnum* {*subname 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 library file entries, it is used in symbol files for device instances to which a name has been assigned with the **Properties** command. The *prefix* can start with any character, but is intended to have significance to SPICE. The character '@' is reserved for the vcc 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. This should always be specified as 0 in the device library file.

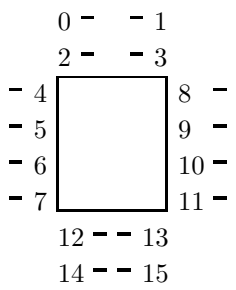
If the name applies to a subcircuit (*not* a subcircuit macro reference in the device library file), the three additional entries may appear in symbol files. The *subname* will be set to "subckt", and the *physX* and *physY* are coordinates where the physical subcircuit instance label is applied. This will be in a physical instance of the subcircuit, or the two coordinates are set to -1 to indicate that the label has not been placed.

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.

B.4.3 Labloc Property: Property Number 12

The name, model, value, and param 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.

Figure B.1: Locations and justification for character position codes around the device bounding box.



`5 12` *pname code* `[` *pname code* `] ...  ;`

The *pname* is one of the literal tokens "`name`", "`model`", "`value`", and "`param`". For backward compatibility, "`initc`" is accepted as an alias for "`param`". The *code* is an integer, -1 – 15. If the *code* is -1, the default placement is used. If code is 0 – 15, the placement and justification are as shown below:

If the height of the device bounding box is greater than the width, the default position codes are name 5, model and value 9, and param 10. Otherwise the default location codes are name 2, model and value 13, and param 14.

B.4.4 Old Mutual Property: 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.

B.4.5 Mutual Property: 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 symbol 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.

B.4.6 Branch Property: 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 A.5).

B.4.7 Labrf Property: Property Number 16

The labrf property is applied by *Xic* to labels that are associated with device properties. The property is not used in the device library file.

```
5 16 name num property;
```

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. 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'.

B.4.8 Mutlrf Property: 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.

B.4.9 Symbolic Property: 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 colon (yes colon!) separated CIF primitives for the symbolic representation, which can include L, B, P, W, and 94 (label) directives.

B.4.10 Nodemap Property: Property Number 19

The nodemap property is applied to the electrical cell definition when node name mapping in SPICE output is to be applied. The property 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 entry can be 0 or nonzero indicating whether node mapping is active by default ("0" turns mapping off). 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 4.14.1 for more information on node mapping.

B.5 Device Properties Set By User

The device properties described in this section provide user-specified information to device instances. These properties can also be applied to device definitions, in which case they provide a default value to the instances of that device.

The name property discussed in the previous section, plus the 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 B.6.

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 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 **Properties** command, and nodes are assigned by *Xic* so as to be unique.

The line looks like:

name n1 ... nLast 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 are numbers or colon separated lists of numbers by default. If the **nodmp** button is active, the nodes may instead be text tokens (see 4.14.1). 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.

B.5.1 Model Property: 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 **Properties** command in 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.

B.5.2 Value Property: Property Number 2

The value property supplies a string to be used in the device line in SPICE output for the device "value". It should not appear if the device has a model property, and if it does, it will be ignored. The property is normally applied to device instances with the **Properties** command, 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.

The model and value properties are mutually exclusive, either can be supplied, but not both.

B.5.3 Parameter Property: 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 **Properties** command, or to cells with the **Cell Properties** 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.

B.5.4 Other Property: 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 **Properties** command. 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;
```

B.5.5 NoPhys Property: 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.

The numerical value of the property is 5. The property string is "`nophys`". *Xic* will always set the property string to this value. Devices inherit this property from cell definitions in the device library file. The format is

```
5 5 nophys;
```

B.6 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 4.11.4). 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 C

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
Paths and Directories	
Path	Symbol file search path
LibPath	Startup file and library search path
HelpPath	Help file search path
ScriptPath	Script file search path
DocsDir	Directory containing release documentation
TeePrompt	Copy messages to given filename or "stdout"
General Visual	
FullWinCursor	Enable full-window cursor
CellThreshold	Min size in pixels of displayed subcell, integer >= 0
LayerTable	Layer table presentation: "`compact`" or "`tiny`"
MaxPrpLabelLen	Max length of displayed prpty string
NoPixmapStore	Don't use screen backing memory
NoDisplayCache	Don't use multi-object rendering for boxes
PhysGridOrigin	Set the origin of the grid displayed in physical mode
'!' Commands	
JoinMaxPolyVerts	Upper bound of vertices in polygons from join (def. 600)
JoinMaxPolyGroup	Limit number trapezoids per poly in join (def. 100)
JoinMaxPolyQueue	Limit number trapezoids to form polys in join (def. 1000)

JoinBreakClean	Manhattan split polygons with too many vertices
LayerPartSize	Partition grid size in microns for layer operations
NoSnapToEdges	Disable edge snapping in object creation commands
Shell	Path to shell used for external commands
SpotSize	Set mask granularity
Selections	
SelectTime	Set delay (msec) to activate move
NoAltSelection	Use legacy click-selection logic
Side Menu Commands	
LogoEndStyle	End style for logos: 0 flush, 1 round, 2 extend
LogoPathWidth	Path width for logos, 1 – 5
LogoManhattan	Use internal Manhattan font for logos
LogoPretty	Use system font for logos
LogoPrettyFont	Name of system font to use for logos
LogoToFile	Create subcell for logos
NoMergeBoxes	No box merging, ever
MaxRoundSides	Maximum sides used to approximate round objects
NoConstrainRound	No DRC constraints creating round objects
PictorialDevs	Use pictorial device menu
ShowDots	Show electrical connections
SPICE Interface	
NoAsyncSpice	Use synchronous *WRspice* interface
SpiceListAll	Include unconnected devices in Spice output
SpiceAlias	Device key aliases for Spice output
SpiceHost	Name of *WRspice* server
SpiceProg	Path name of *WRspice* executable, supersedes below
SpiceExecDir	Directory containing *WRspice* executable
SpiceExecName	Name of *WRspice* executable
CheckSolitary	Report unconnected terminals in netlist
NoSpiceTools	Do not show *WRspice* toolbar
File Menu – Printing	
NoDriverLabels	Don't use driver text for hard copy labels
PSlineWidth	Set line width for postscript line draw
Edit Menu Commands	
MasterMenuLength	Max. entries in master pull-down menu
NoReadExclusive	Don't move stripped path to front of search path
AddToBack	Add stripped path to back of search path
UndoListLength	Number of operations saved in the undo list
NoWireWidthMag	Don't change the width of magnified wires
CrsymOverwrite	Allow Create Cell to overwrite memory cells
View Menu Commands	
InfoInternal	Use internal coordinates in info windows
PeekSleepMsec	Per-layer delay in peek command, milliseconds
XSectThickness	Xsect layer thickness, internal units, deflt 500
RulerSnapToGrid	Rulers snap to grid by default
LockMode	Don't allow physical/electrical mode change
Attributes Menu Commands	
EraseBehindProps	Erase behind phys properties in props command

PhysPropTextSize	Pixel text height used in props command
Convert Menu – General	
NoPopUpLog	Don't pop up log file if warnings or errors
Convert Menu — Input and ASCII Output	
AutoRename	Automatically change clashing cell names when reading
NoAskOverwrite	Suppress prompting for overwrite instructions
NoOverwritePhys	Don't overwrite phys memory cells when reading
NoOverwriteElec	Don't overwrite elec memory cells when reading
MergeInput	Merge boxes and coincident objects when reading
NoPolyCheck	Skip polygon reentrancy tests when reading
NoCoincCheck	Skip coincident object tests when reading
NoCheckEmpties	Skip checking for empty symbols while reading
NoReadLabels	Ignore text labels when reading physical cell data
LayerList	Layer list for conversion input filtering
UseLayerList	How to use layer list, skip or use only
UseLayerAlias	Map layers using layer alias table
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
NoMapDatatypes	New layers take all datatypes in GDSII read
CifLayerMode	CIF layer resolution method, 0–2
SkeletonMode	Use skeleton mode reading archive files
SkelSaveNum	Skeleton mode save geometry threshold
NumToTxtCells	When converting to text, stop after this many cells
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
SkipInvisible	Do not write invisible layers to output
KeepBadArchive	Don't delete failed conversion output archive file
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
CifAddBBox	Add bounding box comment to objects in CIF output
CifOldWire	Output old wire style syntax for backward compatibility
GdsOutLevel	GDSII release level conformance code (0–2)
GdsMunit	Modify M-UNITS value in GDSII output file
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
OasWriteNoGCDcheck	Don't look for common divisors in repetitions
Extract Menu Commands	
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
LayerSearchUp	Iterate layers bottom to top in **paths** command
ExtractOpaque	Ignore the OPAQUE flag in extraction
FlattenPrefix	Cell name prefix to flatten in 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
NoMergeParallel	Never merge parallel devices
NoMergeSeries	Never merge series devices
NoMeasure	Suppress measuring parameters of devices
QpathGroundPlane	**Quick Paths**, use of inverted ground plane, 0–2
NoEnet	Don't print net, **enet** command
EnetSpice	Do include SPICE listing, **enet** command
EnetBottomUp	Use leaf-to-root ordering in electrical netlist
NoPnet	Don't print extracted net list, **pnet** command
NoPnetDevs	Don't print extracted device list, **pnet** command
NoPnetSpice	Don't 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
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
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
NoPermute	Skip permutation search in association
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
FastCap/FastHenry Interface	
FxMinRectSize	Minimum rectangle dimension for Manhattanization
FxPlaneBloat	Overhang of interface planes and dark-field conductors
FxUnits	FastCap/FastHenry file units: m, cm, mm, um, in, mils
FxForeg	FastCap/FastHenry run in foreground if set
FcPartMax	Maximum FastCap top/bottom panel dimension
FcEdgeMax	Maximum FastCap edge panel dimension
FcThinEdge	FastCap outside edge panel width
FcPath	FastCap executable directory path

FcArgs	FastCap command line arguments
FhPath	FastHenry executable directory path
FhArgs	FastHenry command line arguments
FhFreq	FastHenry frequency specification
Help System	
HelpMultiWin	Use separate windows for help references
NoHelpDefault	Suppress default help window

There are a few more variables listed in the table below. These allow setting the colors used peripherally in certain pop-ups and elsewhere. The value is the name of a color.

Variable	Default	Use
pop_color_no	red	Merge Control, Empties, Modified Cells
pop_color_yes	darkgreen	Merge Control, Empties, Modified Cells
pop_color_hl1	red	Debugger, Keys Pressed "INSERT", DRC Rules
pop_color_hl2	darkblue	Modified Cells, Properties
pop_color_hl3	darkviolet	Modified Cells
pop_color_hl4	sienna	DRC Rules, Extract Param, Convert Param, Properties, Cell Properties
pop_color_dvbg	gray90	Pictorial device menu background
pop_color_dvfg	black	Pictorial device menu foreground
pop_color_dvhl	blue	Pictorial device menu highlight
pop_color_dvsl	gray80	Pictorial device menu selection

C.1 Special Constructs

These are special **!set** keywords 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`".

C.2 Paths and Directories

These variables set the search paths (see 1.2.4) and document directory used in *Xic*. These have counterpart environment variables (see 1.2.3). 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 symbol search path. It is always defined, and can not be unset. This path is used to find symbol files and libraries containing symbol files.

If not set by any means, a default symbol 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*`/lib/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*`/lib/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*`/lib/xictools/xic/scripts`)"

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*`/lib/xictools/xic/docs`"

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.

C.3 General Visual

The following **!set** keywords affect general visual attributes of *Xic*.

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.

The supplied **fullcursor** script, which should appear in the **User Menu**, can be used to conveniently toggle this variable.

CellThreshold
Value: integer >= 0.
This sets the size threshold in pixels for 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 **Tiny Boxes** button in the **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.

LayerTable
Value: string, "`compact`" or "`tiny`".
The presentation of the layer table can be altered to squeeze more visible entries into available space by setting this variable to one of the keywords indicated (actually, only the first character of the keyword is significant). If "`compact`", the size of the entry is reduced. If "`tiny`", the size is further reduced, and two rows are presented.

MaxPrpLabelLen
Value: integer >= 6.
This variable sets the maximum width, in default-sized character cells, of a displayed property 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 (32 in releases prior to 2.5.66).

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.

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.

C.4 Keyboard '!' Commands

The **!set** keywords 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.

JoinMaxPolyVerts

Value: integer 0 or 20–8000.
This sets an upper bound on the number of vertices in polygons created with the **!join** command and similar operations. The default is 600 vertices. If set to 0, no limit is applied.

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.
When a collection of trapezoids is being combined into polygons, such as for the **!join** command, 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 100. If set to zero, no limit is applied. Generally, smaller limits allow faster join operations, however this will leave as separate objects more polygons that could have been joined.

JoinMaxPolyQueue

Value: integer >= 0.
When objects are being joined in the **!join** command and similar, 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 (default is 1000) the list is sent to the function that recombines the trapezoids into polygons. This variable is used to set the length threshold. Setting the value to 0 allows the list to grow without bound. Generally, smaller thresholds provide faster processing, but will produce more polygons.

JoinBreakClean

Value: boolean.

In the **!join** command and similar functions, 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.

LayerPartSize

Value: floating-point number.

When geometrical operations are performed over an entire cell, a logical square grid is created over the cell relative to the lower-left corner. The operations are performed for each grid element that intersects the cell area, and the results are combined. This can be more efficient than performing the operations over the entire cell in one shot. Performance rapidly degrades as the amount of geometry per grid area increases. Best performance is 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. If set to "0", gridding is not used.

The **!layer** command and `AdvanceZref` script functions use this variable.

NoSnapToEdges

Value: boolean.

When set, the cursor snapping to edges of existing objects is disabled. This applies to the commands with the following keywords: `polyg`, `round`, `donut`, `arc`, `wire`, `box`, `erase`, `xor`, `break`.

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.

SpotSize

Value: integer 0–1000.

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. Typically, this size is 0.1 to 0.5 microns, 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. *Xic* has features to precondition round objects to avoid this problem: the SpotSize variable and the **!tospot** command.

This variable can be set to the spot size in use, specified in milli-microns. Thus, if the spot size is 0.1 micron, one would use

```
!set SpotSize 100
```

If the SpotSize variable is unset or set to 0, the feature is disabled. The maximum value accepted is 1000. With the SpotSize variable set to 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, 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. This conditioning, with SpotSize set nonzero, applies only to objects created with the **round** and **donut** commands, and not the **arc** command or general polygons.

C.5 Selections

The following **!set** keywords affect object/cell selections using the pointing device.

SelectTime
: **Value:** integer 100–1000.
When button1 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.

C.6 Side Menu Commands

The following **!set** keywords affect the functioning of commands found in the side menu.

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 (the default) or 2 for extended ends.

LogoPathWidth
: **Value:** integer 1–5.
This sets the relative path width used for rendering the vector font in the **logo** command. When set, the prompt for this parameter in the **logo** command is skipped. The variable should be set to an integer 1–5.

LogoManhattan
: **Value:** boolean.
When set, the **logo** command will use an internal bitmap font, and characters will be rendered as Manhattan polygons. This variable can be set from the panel brought up by the **!logofont** command.

LogoPretty

Value: boolean.
When set, the **logo** command will use the system font named in the LogoPrettyFont variable, or a default if this is not set. This variable overrides the LogoManhattan variable. Characters are rendered as Manhattan polygons derived from the font bitmaps. This variable can be set from the panel brought up by the **!logofont** command.

LogoPrettyFont

Value: font name string.
This variable sets the name of a font to be used for text rendering in the **logo** command. This will be used if LogoPretty is set. This variable can be set from the panel brought up by the **!logofont** 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.

LogoToFile

Value: boolean.
If this boolean variable is set, physical text created with the **logo** command will be placed in a file, which is instantiated at the label locations. Otherwise, the physical text is placed directly in the current cell.

NoMergeBoxes

Value: boolean.
When this boolean is set, the automatic merging of boxes is disabled. This also disables box merging in the **Merge In Read** mode.

MaxRoundSides

Value: integer >= 8.
This sets the maximum number of sides that can be used when creating round objects. This allows the user to bypass the built-in default limit when necessary.

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).

PictorialDevs

Value: boolean.
If this boolean is set, the pictorial device menu will be used in electrical mode. The default menu, which occupies less screen space, is a character-keyed pull down menu. The pictorial menu depicts the schematic symbols for each device, and may be more appropriate for new users.

ShowDots

Value: boolean or "a".
This sets the drawing of connection indications in electrical mode, similar to the **dots** button in the side menu (which internally sets this variable). If set to a word starting with "a", all connections are marked with a dot. If set to anything else, only the potentially ambiguous connection points are marked with a dot. When unset, no dots are shown.

C.7 SPICE Interface

The following **!set** keywords affect the interface to the *WRspice* simulator, and SPICE output in general.

NoAsyncSpice
: **Value:** boolean.
When set, the asynchronous interface to *WRspice* is not used during simulations initiated with the **run** command.

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 usually defaults to 3004. If the server uses a non-standard port, and the `spice/tcp` service has not been registered (usually in the `/etc/services` file) on this port, the port number must be provided.

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/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`".

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.

C.8 File Menu — Printing

The following **!set** keywords affect the hard copy drivers.

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.

PSlineWidth
Value: real number 0–25.0.
This variable can be set to a numeric value which will set the linewidth used in the PostScript line-draw mono and color hard copy drivers. One unit is 1/72 inch. The default is 0, which means to use the thinnest line available. This will have no effect in the PostScript bitmap or non-PostScript drivers.

C.9 Edit Menu Commands

The following **!set** keywords affect commands found in the **Edit 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 correct for some screen resolutions. It is not very useful if the pull-down menu extends off-screen.

The treatment of any path which is given with a cell to open in the **Edit** 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 **Edit** 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.

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.

NoWireWidthMag

Value: boolean.
When set, the width of wires does not change when the wire undergoes magnification, in a **Move**, **Copy**, or **Flatten** operation.

CrsymOverwrite
Value: boolean.
When set, The **Create Cell** operation in the **Edit Menu** and the `CreateSymbol` script function can overwrite cells already in memory. This can be dangerous and is prevented by default.

C.10 View Menu Commands

The following **!set** keywords 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 (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.

XSectThickness
Value: integer 100–100000.
This sets the default thickness of the layers as displayed in the **Cross Section** command. The thickness is an integer 100 – 100000 which represents the thickness in internal units (1000 units per micron). If not set, a value of 500 (0.5 microns) is used. If the layer description in the technology file contains a Thickness specification, that value will be used.

RulerSnapToGrid
Value: boolean.
When set, when entering the **Rulers** command the snap-to-grid mode will be initially set.

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.

C.11 Attribute Menu Commands

The following **!set** keywords affect the commands found in the **Attributes Menu**.

EraseBehindProps
Value: boolean.
If given, the area inside the text bounding box of text generated by the **Show Properties** command is erased, to promote visibility of the text.

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 with the **Show Properties** command. If not set, the default is 14.

C.12 Convert Menu — General

Below are variables relating to data format translation, both reading and writing.

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 **Edit** 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.

C.13 Convert Menu — Input and ASCII Output

The **!set** keywords below affect the format conversion when importing data from a file. Many of these variables have counterpart buttons in the **Conversion – Import** panel from the **Convert Menu**. The functionality also applies in many cases when input is being read in the **Edit** command and similar.

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 9.1).

NoAskOverwrite (boolean)
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.

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.

MergeInput
Value: boolean.
When this variable is set, boxes are merged together, if possible, as files are being read in. Overlapping boxes are clipped and/or merged. Objects (not subcells) that are coincident with an identical object will be deleted. This applies to all geometrical objects and not just boxes. Merging applies to conversions as well as cells being read with the **Edit** command.

The merging of boxes is suppressed if the NoMergeBoxes variable is set. Coincident boxes will still be deleted, however.

NoPolyCheck
Value: boolean.
When this boolean variable is set, the reentrancy test for polygons is skipped while an input file is being read. The default behavior is to check each polygon for potentially troublesome geometry specification while the polygon is being created.

NoCoincCheck
Value: boolean.
When set, testing for coincident objects (but not subcells) is skipped while reading input. Coincident subcells are always reported in the log file. This has no effect on the merging of coincident objects if the MergeInput variable is set.

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 symbols in the hierarchy is suppressed. An "empty symbol" is a symbol that contains no physical or electrical objects or subcells. It is possible to check for empty symbols at any time with the **!empties** command.

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. In multi-pass read operations, such as skeleton mode and when using the hierarchy context structure, the variable state should be the same when reading the initial context as when reading geometry, otherwise cell bounding boxes may not be accurate during the geometry read operation.

LayerList
Value: string.
This can be set to a space-separated list of layer names (see 9.3). 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 tracks the entry areas in the **Conversion – Import** and **Conversion** panels. 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. The layer list is used, if enabled by this variable, when

- Input is being read from an archive file.
- The operation was initiated from the **Conversion – Import** or **Conversion** panel, or from one of the `FromArchive`, `Edit`, or `OpenFile` script functions. The LayerList is ignored otherwise, whether of not UseLayerList is set.

- The physical-mode part of the file is being read. The LayerList applies only to physical data.

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.

UseLayerAlias

Value: boolean.
When this variable is set, when reading an archive or native file, layers encountered are aliased according to entries in an internal table of aliases. This table is manipulated with the **Layer Alias Editor** available in the **Conversion − Import** and **Conversion** panels, or with script functions, or with the **!lal** command. 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.

InToLower

Value: boolean.
When set, symbol 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 9.1).

InToUpper

Value: boolean.
When set, symbol 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 9.1).

InUseAlias

Value: boolean or string.
This variable enables utilization of the alias file (see 9.2) 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 9.1).

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 9.1).

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.

CifLayerMode
Value: integer 0–2.
This variable determines how *Xic* interprets layer directives while reading CIF files. This is the same as the layer resolution option menu in the **Conversion − Import** panel. Setting to 0 is the default **Try Both** option, 1 is the **By Name** option, and 2 is the **By Index** option.

SkeletonMode
Value: boolean.
When set, archive files will be read in "skeleton" mode. In this mode, the cell hierarchy but not the geometry is saved in memory. Geometry is accessed directly from the file (and cached) when needed. Skeleton mode is described in 9.12.

SkelSaveNum
Value: integer >= 0.
This controls a feature in skeleton mode that can improve processing speed. If a cell contains this many geometrical objects or fewer, it is retained in memory. The assumption is that there are small cells that are instantiated in many places, and it is more efficient to cache these cells rather than having to update from the file each time the content is needed. This variable should be set to some small number. If set to 0, this feature is effectively disabled.

NumToTxtCells
Value: integer.
When converting GDSII or CGX to a text-mode representation file, writing will stop after the number of cells given in this variable have been processed. Setting to zero will show only the file header records. If negative or the variable is unset, the entire file will be written. In the case of GDSII, such files are not back-convertible to GDSII.

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 **Converson** 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 **Converson** panel.

C.14 Convert Menu — Output

The **!set** keywords below affect the format conversion when writing data to a file. Many of these variables have counterpart buttons in the **Conversion – Export** panel from the **Convert menu**. The functionality may also apply to files created with the **Save** command and similar.

StripForExport
: **Value:** boolean.
When this variable is set, files produced through the **Conversion – Export** panel, or with the `ToCGX`, `ToCIF`, `ToGDS`, or `ToOASIS` script functions, will contain the basic syntax elements with no extensions. Thus, they contain physical data only. See the description of the counterpart button in the **Conversion – Export** panel for complete information.

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 using the conversions initiated from the **Conversion - Export** panel, or with the `ToCGX`, `ToCIF`, `ToGDS`, `ToOASIS`, or `ToXIC` script functions. 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.

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.

OutToLower
: **Value:** boolean.
When set, symbol 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 9.1).

OutToUpper
: **Value:** boolean.
When set, symbol 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 9.1).

OutUseAlias
: **Value:** boolean or string.
This variable enables utilization of the alias file (see 9.2) 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 9.1).

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 9.1).

CIFoutStyle

Value: string.
When set, this variable will determine the CIF output style. Changing the option menu choices in the **Conversion − Export** pop-up will update the value of CIFoutStyle, but only if the variable is already set. If the variable is not set, reading in a CIF file will set the internal style parameters, so that if the file is written, it will have the same style. Before writing, the style can be changed from the **Conversion − Export** pop-up.

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 **Conversion − Export**.

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 **Conversion − Export** pop-up while CIFoutStyle is set, the value of CIFoutStyle will have this form.

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*`,`*top width height*`);`

CifOldWire

Value: boolean.

When set, CIF output (without the **Strip For Export** button set), and the native symbol files, will use the old syntax for wires, which is compatible with older releases of *Xic* (before 2.5.23). It is necessary to use this option when exporting files to be read by older releases if the cells contain wires with non-default end style. The new syntax is more efficient and should be used if possible.

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*).

The **Conversion – Export** panel from the **Convert Menu** has an option menu for effectively setting the GdsOutLevel variable.

GdsMunit

Value: real 0.01 – 100.0.
When writing GDSII, the normal MUNITS value (1e-9) is multiplied by this factor when written to the physical mode part of the file. This should be accompanied by a compensating scale factor change in order to keep the file scaling constant, unless a change of scale is desired. The acceptable range is 0.01 – 100.0. This will apply to *all* GDSII files written.

NoGdsMapOk

Value: boolean.
When this boolean 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.

OasWriteCompressed

Value: boolean.
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. This applies to OASIS files generated from the **Convert** and **File** menus and equivalent functions. The variable tracks the setting of the check box in the **Conversion – Export** and **Conversion** panels.

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).

The variable tracks the setting of the check box in the **Conversion – Export** and **Conversion** panels.

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 may 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*]

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 three. 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 two. There can be no space around the '=', and *N* must be an integer. This is ignored if r is given.

If OasWriteRep is set to an empty string, all objects will be processed for replication, using the default run and array minimums.

The variable tracks the setting of the check box in the **Conversion** and **Conversion – Export** panels, which will effectively set/unset this variable *as a boolean*. Thus, to use repetitions in other than the default mode, this variable must be set explicitly to the appropriate value as described above.

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 **Conversion – Export** and **Conversion** panels. 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.

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.

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.

C.15 Extraction Menu Commands

The **!set** keywords below affect the commands found in the **Extract 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.

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.

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 20.

LayerSearchUp
Value: boolean.
When set, the layers are searched from bottom to top in the extraction functions. Otherwise, layers are searched from top to bottom. This affects the association of terminals to layers, and the layer selected in the **Show Paths** and **Quick Paths** commands in the **Extract Menu**.

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.

FlattenPrefix
Value: string.
In extraction, physical subcells whose name begins with the string value of this variable are flattened into the parent cell for extraction purposes.

Note: in releases of *Xic* prior to 2.5.19, this variable was named "PnetFlattenPrefix".

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.

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.

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 11.1.1).

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.

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.

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.

QpathGroundPlane

Value: integer 0–2.

This variable controls how the **Quick Paths** command uses the inverted ground plane. An inverted ground plane is created while grouping under extraction, if enabled. The choices are

0 Use the inverted ground plane if available
1 Create the inverted ground plane if necessary, and use it
2 Never use the inverted ground plane

The default behavior is to use the inverted ground plane if it already exists, as for "0", if this variable is unset.

NoEnet

Value: boolean.

If set, the netlist is skipped when writing output in the **Dump Elec Netlist** command. This variable corresponds to the net check box available in that command, with inverse logic.

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.

NoPnet
Value: boolean.
If set, the extracted netlist listing is skipped in output from the **Dump Phys Netlist** command. This variable corresponds to the net check box available in that command, with inverse logic.

NoPnetDevs
Value: boolean.
If set, the extracted device listing is skipped in output from the **Dump Phys Netlist** command. This variable corresponds to the devs check box available in that command, with inverse logic.

NoPnetSpice
Value: boolean.
If set, the SPICE listing of extracted devices is skipped in output from the **Dump Phys Netlist** command. This variable corresponds to the spice check box available in that command, with inverse logic.

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.

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. The variable is also used by the **!source** 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. The variable is also used by the **!source** 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. The variable is also used by the **!source** command.

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. The variable is also used by the **!exset** command.

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. The variable is also used in the **!exset** command.

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. The variable is also used in the **!exset** 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. The variable is also used in the **!exset** command.

NoPermute

Value: boolean.

When this variable is set, the association algorithm will not attempt to iterate through the combinations when searching for a solution. Many circuits do not require a permutation search, however some circuits, and in particular circuits where the wiring is incomplete, may require a lot of time

for the permutation search. Of course, if a permutation search is needed and not performed, LVS will fail. This variable is mostly for debugging, or for cases where association is not needed.

Permutes are also skipped if a device or subcircuit is found that can not possibly be associated.

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.

C.16 FastCap/FastHenry Interface

The following variables apply to the FastCap/FastHenry interface. Most of these are associated with entry fields in the **RLC Extraction** panel, which is brought up with the **Extract RLC** button in the **Extract Menu**.

FxMinRectSize

Value: real number 0.05–10.0.
When a non-Manhattan polygon is added to the **RLC Extraction** database with the **Save Selections** button or equivalent, 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. The value is specified in microns. If not set, a value of 1.0 microns is used. This variable is most conveniently manipulated with the text input field in the **RLC Extraction** panel.

FxPlaneBloat

Value: real number 0.0–100.0.
Reverse-polarity conductors in FastCap and FastHenry, and interface planes in FastCap, are extended beyond the bounding box of the conductor patterning by a distance that can be specified with this variable. This geometry is generated internally. If not set, the value used is 10.0 microns.

This variable is most conveniently manipulated with the text input field in the **RLC Extraction** panel.

FxUnits

Value: units string.
This variable can be used to specify the length units used in generated FastCap and 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 **RLC Extraction** panel.

FxForeg

Value: boolean.
If this variable is set, as a boolean, the **Extract C** and **Extract RL** buttons in the **Extract RLC** panel will initiate a FastCap or 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.

FcPartMax

Value: real number 0.05–1000.0.
This is the maximum dimension of a FastCap top/bottom surface partition panel. All panels are limited to this value during the initial partitioning. The value given is in microns, in the range 0.05 – 10.0. If not set the value used is 10.0. This variable is most conveniently manipulated with the text field in the **RLC Extraction** panel.

FcEdgeMax

Value: real number 0.05–1000.0.
This sets the maximum length of a conductor edge "vertical" partition in FastCap. The value given is in microns, in the range 0.05 – 10.0. If not set, the value used is 10.0. This variable is most conveniently manipulated with the text field in the **RLC Extraction** panel.

FcThinEdge

Value: real number 0.01–1.0.
This is the width of a special thin partition created in the top/bottom surface along the outside edges of conductors for FastCap. The value given is in microns, in the range 0.05 – 1.0. If not set, the value used is 0.5 microns. This variable is most conveniently manipulated with the text field in the **RLC Extraction** panel.

FcPath

Value: directory path string.
This variable can be set to a path to a directory containing the `fastcap` binary. If not set, if FastCap is being run in the background, the `fastcap` executable is expected to be found in the same directory as the `wrspice` executable (see the SpiceExecDir variable description in C.7). If running in the foreground, the current shell search path will be used to locate the executable. If this variable is set, the executable will be searched for in the specified directory only.

FcArgs

Value: string.
This variable can be set to a string, which will be included in the argument list when FastCap is initiated with the **Extract C** button in the **Extract RLC** panel. The variable is most conveniently manipulated with the text entry field in the **RLC Extraction** panel.

FhPath

Value: directory path string.
This variable can be set to a path to a directory containing the `fasthenry` binary. If not set, if

FastHenry is being run in the background, the `fasthenry` executable is expected to be found in the same directory as the `wrspice` executable (see the SpiceExecDir variable description in C.7). If running in the foreground, the current shell search path will be used to locate the executable. If this variable is set, the executable will be searched for in the specified directory only.

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 **Extract RL** button in the **Extract RLC** panel. The variable is most conveniently manipulated with the text entry field in the **RLC Extraction** panel.

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 **RLC Extraction** panel.

C.17 Help System

The following **!set** keywords affect the help system.

HelpMultiWin

Value: boolean.
This variable, when set, causes the help system to use a new window for each selected reference. By default, the original help window is reused.

NoHelpDefault

Value: boolean.
If this variable is set, the default help window which normally appears when the **Help** button is pressed is suppressed. Help can be obtained in the usual way by pressing buttons or through other actions.

This page intentionally left blank.

Appendix D

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 (rather lengthly) table that follows lists the interface functions presently available, by category. These are described in detail in the following sections. 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.

Control Functions	
Current Cell	
`Edit(`*name*`, `*symname*`)`	Edit cell
`OpenCell(`*name*`, `*symname*`)`	Read file into memory
`NewCellName()`	Return empty new cell name
`ClearCell(`*undoable*`, `*layer_name*`)`	Clear content of current cell
`Push(`*object_handle*`)`	Push editing context to subcell
`Pop()`	Pop editing context to parent
`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
`Save(`*newname*`)`	Save to disk
`UpdateNative(`*dir*`)`	Save modified hierarchy cells as native
Database	
`RenameCell(`*oldname*`, `*newname*`)`	Globally rename cell in memory, fix references
`Clear(`*name*`)`	Delete cells from memory
`ClearAll()`	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()`	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
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
`ShowSymbolic(`*show*`)`	Turn on/off symbolic display
Exit	
`Exit()`	Exit script
`Halt()`	Exit script
Annotation	
`AddMark(`*type*`, `*arguments ...*`)`	Show a user-specified mark
`EraseMark(`*id*`)`	Erase a mark
Ghost Rendering	
`PushGhost(`*array*`, `*numpts*`)`	Register ghost-drawn polygon
`PushGhostBox(`*left*`, `*bottom*`, `*right*`, `*top*`)`	Register ghost-drawn box
`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*, *radius*, *theta1*, *theta2*)	Draw arc
GRpolygon(*handle*, *array*, *num*)	Draw polygon
GRtext(*handle*, *text*, *x*, *y*, *transform*)	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
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
Libraries	
OpenLibrary(*path_name*)	Open a library file
CloseLibrary(*path_name*)	Close an open library
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
Technology File	
Techfile(*techname*)	Not implemented
GetTechExt()	Return technology file extension
SetTechExt(*extension*)	Define effective technology file extension
Variables	
Set(*name*, *string*)	Set a variable
Unset(*name*)	Unset a variable
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
Utility Functions	
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
Cell Name Enumeration	
InitGen()	Return handle to subcell name list
CellsHandle(*cellname*, *depth*)	Return handle to subcell name list
GenCells(*handle*)	Return name from name list
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*)	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
Miscellaneous	
SepString(*string*, *repeat*)	Create separation or indentation string
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
ToReal(*string*)	Convert string to number
ToString(*real*)	Convert number to string
ToFormat(*format*, *arg_list*)	Print variables according to format string
ToChar(*integer*)	Convert character constant to string representation
String and Array Size	
Strlen(*string*)	Return length of string
Sizeof(*arg*)	Return string length or array size
Operating System and Input/Output Functions	
Archive File Context	
FileInfo(*filename*, *handle_or_filename*, *flags*)	Obtain info about archive file
GetArchiveContext(*filename*, *scale*, *save_pcinfo*)	Create new archive context
SaveArchiveContext(*cx_handle*, *name*)	Save archive context
RecallArchiveContext(*name*, *remove*)	Retrieve archive context
WriteArchiveContext(*cx_handle*, *filename*)	Write archive context to file
ReadArchiveContext(*filename*)	Obtain archive context from file
CxInfo(*cx_handle*, *handle_or_filename*, *flags*)	Obtain archive context information
CxEstFlatMemoryUse(*cx_handle*, *cellname*, *array*, *counts_array*)	Estimate memory required for flat read
CxCompare(*cx_handle1*, *cx_handle2*, *cellname*, *layer_list*, *obj_types*, *skip_layers*, *geometric*, *array*)	Compare objects in cell between archives
CxFileName(*cx_handle*)	Obtain archive file name
CxFileType(*cx_handle*)	Obtain archive file format
CxTopCells(*cx_handle*)	Obtain archive top-level cell names
CxLayers(*cx_handle*)	Obtain layers used in archive
CxEstSkelSize(*cx_handle*)	Obtain estimated skeleton memory requirement
CxCellBB(*cx_handle*, *cellname*, *array*)	Obtain cell bounding box
CxSetArea(*cx_handle*, *cellname*, *l*, *b*, *r*, *t*)	Configure context to area

CxClearArea(*cx_handle*)	Undo area configuration
CxSetSkipFlag(*cx_handle*, *cellname*, *skip*)	Set or clear skip flag
CxSetSkipEmpties(*cx_handle*, *cellname* [, *left*, *bottom*, *right*, *top*])	Set skip flags for empty cells
CxClearSkipFlags(*cx_handle*)	Unset all skip flags
CxEdit(*cx_handle*, *cellname*)	Open cell for editing
CxSetupSkeleton(*cx_handle*, *cellname*, *l*, *b*, *r*, *t*)	Create skeleton-mode cells
CxWrite(*cx_handle*, *scale*, *cellname*, *array*, *clip*, *all*, *flatten*, *no_empties*, *outfile*)	Write cells to file
CxOpenFlat(*cx_handle*, *scale*, *cellname*, *array*, *clip*)	Read a flattened hierarchy into memory
Conversion	
SetConvertScale(*scale*, *which*)	Set scale factor for import/export
SetMapToLower(*state*, *rw*)	Set cell name case conversion
SetMapToUpper(*state*, *rw*)	Set cell name case conversion
FromArchive(*archive_file*, *destination*)	Translate archive file to another format
FromTxt(*text_file*, *gds_file*)	Create GDSII file from GDSII text
Conversion — Export	
SaveCellAsNative(*cellname*, *directory*)	Write a native cell file in the directory
SetStripForExport(*state*)	Set flag to write physical data only
SetSkipInvisLayers(*code*)	Set code to skip invisible layers in output
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*, *symbol_list*)	Write GDSII library file
ToOASIS(*oas_name*)	Write OASIS file
ToTxt(*archive_file*, *text_file*)	Write text-mode GDSII file
ToSpice(*spicefile*)	Write SPICE file
WriteGeomSplit(*fname*, *cname*, *bname*, *l*, *b*, *r*, *t*, *x0*, *y0*, *w*, *h*, *bw*)	Convert to flat files
Conversion — Import	
SetConvertFlags(*use_window*, *clip*, *flatten*, *no_empties*)	Set modes for format translation
SetConvertArea(*l*, *b*, *r*, *t*)	Set filter/clipping area for translation
SetMergeInRead(*state*)	Enable box and wire merging in input
Skeleton Mode	
SkelSetDataWin(*l*, *b*, *r*, *t*)	Set skeleton mode clipping window
SkelClear()	Clear or revert skeleton hierarchies
SkelClearGeom()	Clear skeleton mode geometry cache
SkelReadGeom(*depth*)	Load skeleton mode geometry cache
SkelWriteGeom(*depth*, *filename*, *flat*, *clip*)	Write data to file
SkelWriteGeomSplit(*basename*, *x0*, *y0*, *wid*, *hei*, *border*)	Write flattened data to files
SkelWriteDigest(*digest_file*)	Write digest file for current cell/archive
SkelReadDigest(*digest_file*, *archive_file*)	Read digest file and set current cell
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_context*`,` *scale*`,` *layer_list*`,` *use_layer_list*`,` *layer_aliases*`,` *cname_prefix*`,` *cname_suffix*`,` *ccvt*`)`	Register a source archive for streaming
`StreamInstance(`*stream_handle*`,` *cellname*`,` *x*`,` *y*`,` *my*`,` *rot*`,` *magn*`,` *no_empties*`,` *flatten*`,` *array*`,` *clip*`)`	Add an instance conversion spec to a source
`StreamRun(`*stream_handle*`)`	Initiate streaming to output
Current Directory	
`Cwd(`*path*`)`	Set current directory
`Pwd()`	Return current directory
Date and Time	
`DateString()`	Return the date/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(`*directory*`,` *filter*`)`	Return handle to list of file names
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
System Command Interface	
`Shell(`*command*`)`	Execute a shell command
`System(`*command*`)`	Execute a shell command
`GetPID(`*parent*`)`	Return process ID
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	
`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
Geometry Management Functions	
General	
`Commit()`	Finalize changes in database
`Undo()`	Undo last operation
`Redo()`	Redo last undone operation
Current Transform	
`SetTransform(`*angle*`, `*reflection*`, `*magnification*`)`	Set current transform
`StoreTransform(`*register*`)`	Save current transform parameters
`RecallTransform(`*register*`)`	Recall current transform parameters
`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
Grid	
`ShowGrid(`*on*`, `*win*`)`	Set grid visibility in window
`ShowAxes(`*style*`, `*win*`)`	Set axes style in window
`SetGrid(`*interval*`, `*snap*`, `*win*`)`	Set grid parameters for window
`GetGridInterval(`*win*`)`	Return grid spacing
`GetGridSnap(`*win*`)`	Return grid snap number
`SetGridStyle(`*style*`, `*win*`)`	Set grid line style
`GetGridStyle(`*win*`)`	Return grid line style
`SetGridOnTop(`*ontop*`, `*win*`)`	Set grid on top of geometry
`GetGridOnTop(`*win*`)`	Return grid top/bottom status
`ClipToGrid(`*coord*`, `*win*`)`	Move coord to grid
Layers	
`SetCurLayer(`*name*`)`	Set current layer, layer must exist
`SetCurLayerAlias(`*longname*`)`	Set long name of current layer
`SetCurLayerDescr(`*description*`)`	Set description of current layer
`SetCurLayerExKeyword(`*string*`)`	Set extraction keyword/value of current layer
`RemoveCurLayerExKeyword(`*keyword*`)`	Remove extraction keyword spec from current layer
`NewCurLayer(`*name*`)`	Set current layer, create if necessary
`GetCurLayer()`	Return name of current layer
`GetCurLayerAlias()`	Return long name of current layer

`GetCurLayerDescr()`	Return description of current layer
`AddLayer(`*layer_name*`)`	Add a new layer
`RemoveLayer(`*layer_name*`)`	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
`IsLayerConductor(`*lname*`)`	Return nonzero for Conductor
`IsLayerRouting(`*lname*`)`	Return nonzero for Routing
`IsLayerGround(`*lname*`)`	Return nonzero for GroundPlane
`IsLayerContact(`*lname*`)`	Return nonzero for Contact
`IsLayerVia(`*lname*`)`	Return nonzero for Via
`IsLayerDarkField(`*lname*`)`	Return nonzero for DarkField
`IsLayerVisible(`*lname*`)`	Return nonzero for Visible
`SetLayerVisible(`*lname*`, `*visible*`)`	Set Visible flag
`GetLayerMinDimension(`*lname*`)`	Return minimum dimension
`GetLayerWireWidth(`*lname*`)`	Return default wire width
`GetLayerThickness(`*lname*`)`	Return Thickness
`GetLayerRho(`*lname*`)`	Return resistivity
`GetLayerEps(`*lname*`)`	Return dielectric constant
`GetLayerLambda(`*lname*`)`	Return penetration depth
`GetLayerResis(`*lname*`)`	Return resistance per square
`GetLayerCap(`*lname*`)`	Return capacitance per area
`GetLayerCapPerim(`*lname*`)`	Return capacitance per length
`AddLayerGdsOutMap(`*lname*`, `*layer_num*`, `*datatype*`)`	Add GDSII output layer mapping
`RemoveLayerGdsOutMap(`*lname*`, `*layer_num*`, `*datatype*`)`	Remove GDSII output layer mapping
`AddLayerGdsInMap(`*lname*`, `*string*`)`	Add GDSII input layer mapping
`ClearLayerGdsInMap(`*lname*`)`	Clear GDSII input layer mapping
`SetLayerNoDRCdatatype(`*lname*`, `*datatype*`)`	Set GDSII NoDRC datatype
Layer Alias Table	
`ReadLayerAliases(`*handle_or_filename*`)`	Read file containing layer aliases
`DumpLayerAliases(`*handle_or_filename*`)`	Dump file containing layer aliases
`ClearLayerAliases()`	Delete all layer aliases
`AddLayerAlias(`*lname*`, `*new_lname*`)`	Add layer alias to table
`RemoveLayerAlias(`*lname*`)`	Remove layer alias from table
`LayerAlias(`*lname*`)`	Return alias for layer name
Selections	
`SetLayerSpecific(`*state*`)`	Set layer-specific mode
`SetSelectMode(`*ptr_mode*`, `*area_mode*`, `*sel_mode*`)`	Set selection modes
`Select(`*left*`, `*bottom*`, `*right*`, `*top*`, `*types*`)`	Select objects
`SelectLast(`*types*`)`	Select most recent new object
`Deselect()`	Deselect objects
Object Management by Handles	
`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
`MakeObjectCopy(`*numpts*`, `*array*`)`	Create a phony object copy
`BloatObjects(`*object_handle*`, `*dimen*`, `*lname*`, `*mode*`)`	Create list of bloated objects
`EdgeObjects(`*object_handle*`, `*dimen*`, `*lname*`, `*all*`)`	Create list of edge "wire" polygons
`ManhattanizeObjects(`*object_handle*`, `*dimen*`, `*lname*`)`	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*`, `*lname*`, `*vert*`)`	Split into trapezoids objects in a list
`CheckObjectsConnected(`*object_handle*`)`	Return 1 if objects in list form one group
`CheckForHoles(`*object_handle*`, `*all*`)`	Return 1 if object(s) have "holes"
`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
`ObjectDelete(`*object_handle*`)`	Delete an object
`ObjectSelect(`*object_handle*`)`	Select an object
`ObjectDeselect(`*object_handle*`)`	Deselect a selected object
`ObjectMove(`*object_handle*`, `*refx*`, `*refy*`, `*x*`, `*y*`, `*all*`)`	Move object(s)
`ObjectMoveToLayer(`*object_handle*`, `*refx*`, `*refy*`, `*x*`, `*y*`, `*all*`, `*oldlayer*`, `*newlayer*`)`	Move object(s) with layer change
`ObjectCopy(`*object_handle*`, `*refx*`, `*refy*`, `*x*`, `*y*`, `*all*`)`	Copy object(s)
`ObjectCopyToLayer(`*object_handle*`, `*refx*`, `*refy*`, `*x*`, `*y*`, `*all*`, `*oldlayer*`, `*newlayer*`)`	Copy object(s) with layer change
`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*`, `*touchok*`, `*useall*`, `*remove*`)`	Select objects via template
`FilterObjectsA(`*object_list*`, `*array*`, `*array_size*`, `*touchok*`, `*remove*`)`	Select objects via given polygon
`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
`GetObjectBB(`*object_handle*`, `*array*`)`	Return the object's bounding box
`SetObjectBB(`*object_handle*`, `*array*`)`	Set the object's bounding box, scale object
`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
`GetObjectState(`*object_handle*`)`	Return the object's state
`GetObjectGroup(`*object_handle*`)`	Return the object's conductor group
`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
GetLabelXform(*object_handle*)	Return transform code for label
SetLabelXform(*object_handle*, *xform*)	Set transform code 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
GetInstanceName(*object_handle*)	Return name of instance cell
SetInstanceName(*object_handle*, *newname*)	Set instance name, replace instance
Pseudo-Flat Generator Functions	
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
Clipping Functions	
ClipAround(*object_handle1*, *object_handle2* [, *all1*, *all2*])	Clip object around other objects
ClipAroundCopy(*object_handle1*, *object_handle2* [, *lname*, *all1*, *all2*])	Clip objects around other objects, return copies
ClipAroundAll(*object_handle1*, *object_handle2*)	Clip object list around another object list
ClipAroundAllCopy(*object_handle1*, *object_handle2*)	Clip object list around another object list, return copies
ClipTo(*object_handle1*, *object_handle2* [, *all1*, *all2*])	Clip objects to other objects
ClipToCopy(*object_handle1*, *object_handle2* [, *lname*, *all1*, *all2*])	Clip objects to other objects, return copies
ClipToAll(*object_handle1*, *object_handle2*)	Clip object list to another object list
ClipToAllCopy(*object_handle1*, *object_handle2*)	Clip object list to another object list, return copies
ClipObjects(*object_handle*, *merge*)	Clip object list so no overlap
ClipIntersectCopy(*object_handle1*, *object_handle2*, *lname*, *all1*, *all2*)	Exclusive-or objects or lists
Other Object Management Functions	
ChangeLayer()	Change layer of selected objects
Bloat(*dimen*, *mode*)	Bloat selected objects
Manhattanize(*dimen*)	Manhattanize selected objects
Join()	Join selected objects

`Decompose(`*vert*`)`	Convert selected objects to trapezoids
`Box(`*left*`, `*bottom*`, `*right*`, `*top*`)`	Create a box
`Polygon(`*num*`, `*arraypts*`)`	Create a polygon
`Arc(`*x*`, `*y*`, `*rad1X*`, `*rad1Y*`, `*rad2X*`, `*rad2Y*`, `*ang_start*`, `*ang_end*`)`	Create an arc polygon
`Sides(`*numsides*`)`	Set the number of sides used for round objects
`Wire(`*width*`, `*num*`, `*arraypts*`, `*end_style*`)`	Create a wire
`Label(`*text*`, `*x*`, `*y*` [, `*width*`, `*height*`, `*xform*`])`	Create a label
`Logo(`*string*`, `*x*`, `*y*` [, `*width*`, `*height*`])`	Create physical text
`Justify(`*hj*`, `*vj*`)`	Set default text justification
`Place(`*cell*`, `*x*`, `*y*`)`	Place an instance
`PlaceTemplateArgs(`*cell*`, `*args*`)`	Set template parameter string
`Replace(`*cellname*`, `*add_xform*`)`	Replace an instance
`PlaceParams(`*llref*`, `*nx*`, `*ny*`, `*dx*`, `*dy*`)`	Set default placement parameters
`Delete()`	Delete selected objects
`DeleteEmpties(`*recurse*`)`	Delete empty cells
`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*`)`	Copy selected objects
`CopyToLayer(`*fromx*`, `*fromy*`, `*tox*`, `*toy*`, `*oldlayer*`, `*newlayer*`)`	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*`)`	Rotate selected objects
`RotateToLayer(`*x*`, `*y*`, `*ang*`, `*oldlayer*`, `*newlayer*`)`	Rotate selected objects and change layer
`Split(`*x*`, `*y*`, `*flag*`, `*orient*`)`	Divide selected objects
`Flatten(`*depth*`, `*fast*`)`	Flatten hierarchy
`CreateSymbol(`*symname*`, [`*orig_x*`, `*orig_y*`])`	Create new symbol from selected objects
`Layer(`*string*`, `*depth*`, `*flat*`, `*undoable*`)`	Apply geometric manipulations
Property Management by Handles	
`PrptyHandle(`*object_handle*`)`	Return handle to a list of the object's properties
`PrptyNumber(`*prpty_handle*`)`	Return the property number
`PrptyString(`*prpty_handle*`)`	Return the property string
`PrptyNext(`*prpty_handle*`)`	Advance to the next property
`PrptyAdd(`*object_handle*`, `*number*`, `*string*`)`	Add a property
`PrptyRemove(`*object_handle*`, `*number*`, `*string*`)`	Remove a property
`GetPrpHandle(`*number*`)`	Return a handle to certain properties
`GetCellPrpHandle(`*number*`)`	Return a handle to list of current cell properties
`GetProperty(`*prpty_handle*`, `*number*`)`	Get the next property from a properties list
`GetPropertyString(`*object_handle*`, `*number*`)`	Get a property string
Other Property Management Functions	
`AddProperty(`*number*`, `*string*`)`	Add properties to selected objects
`AddCellProperty(`*number*`, `*string*`)`	Add property to current cell

RemoveProperty(*number*, *string*)	Remove properties from selected objects
RemoveCellProperty(*number*, *string*)	Remove properties from current cell
Design Rule Checking	
DRCstate(*state*)	Set interactive DRC
DRCsetLimits(*batch_cnt*, *intr_cnt*, *intr_time*, *skip_cells*)	Set DRC limit values
DRCgetLimits(*array*)	Return DRC limit values
DRCsetLevel(*level*)	Set DRC error reporting level
DRCgetLevel()	Return DRC error reporting level
DRCcheckArea(*left*, *bottom*, *right*, *top*, *file_handle*)	Perform DRC in area
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
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
Zoidlists and Layer Expressions	
SetZref(*thing*)	Set background clipping zoidlist
GetZref()	Return background clipping zoidlist
AdvanceZref(*clear*)	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(*layername*, *depth*)	Create zoidlist from cell
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)
TransformZ(*zoid_list*, *refx*, *refy*, *newx*, *newy*)	Apply a transformation to a zoidlist
BloatZ(*dimen*, *zoid_list*, *mode*)	Bloat a zoidlist
EdgesZ(*dimen*, *zoid_list*, *mode*)	Create an edge zoidlist
ManhattanizeZ(*dimen*, *zoid_list*)	Manhattanize a zoidlist
RepartitionZ(*zoid_list*)	Canonicalize for horizontal split

`ParseLayerExpr(`*string*`)`	Create lexper from string
`EvalLayerExpr(`*lexper*`, `*zoidlist*`, `*depth*`, `*isclear*`)`	Evaluate layer expression in zoidlist
`TestCoverage(`*lexper*`, `*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*`)`	Save trapezoid list in file
`ZfromFile(`*filename*`)`	Extract trapezoid list from file
`CxGetZlist(`*cx_handle*`, `*cellname*`, `*scale*`, `*array*`, `*clip*`, `*all*`)`	Extract trapezoid list from context
`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
`ClearSPtable(`*name*`)`	Destroy a table
`GetSPdata(`*name*`, `*x*`, `*y*`)`	Obtain value from table
`SetSPdata(`*name*`, `*x*`, `*y*`, `*value*`)`	Set table value
Polymorphic Flat Database	
`CxOpenOdb(`*cx_handle*`, `*scale*`, `*cellname*`, `*array*`, `*clip*`, `*dbname*`)`	Open a flat object database
`CxOpenZdb(`*cx_handle*`, `*scale*`, `*cellname*`, `*array*`, `*clip*`, `*dbname*`)`	Open a flat trapezoid database
`CxOpenZbdb(`*cx_handle*`, `*scale*`, `*cellname*`, `*array*`, `*clip*`, `*dbname*`)`	Open a binned flat trapezoid database
`GetObjectsOdb(`*dbname*`, `*layer_list*`, `*array*`)`	Read objects from database
`GetZlistDb(`*dbname*`, `*layer_name*`, `*zoidlist*`)`	Read trapezoids from database
`ShowDb(`*dbname*`, `*array*`)`	Display database region
`DestroyDb(`*dbname*`)`	Destroy a database
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
`ExtractRL(`*conductor_zoidlist*`, `*layername*`, `*r_or_l*`, `*array*`, `*term*`, ...)`	Extract resistance or inductance from object
`ExtractNetResistance(`*x*`, `*y*`, `*spicefile*`, `*array*`, `*term*`, ...)`	Extract resistance from wire net
Terminals	
`ModifyTerminal(`*xe*`, `*ye*`, `*xp*`, `*yp*`, `*name*`, `*lname*`, `*type*`, `*remove*`)`	Modify connection point

GetTerminalName(*terminal_handle*)	Return terminal name
GetTerminalType(*terminal_handle*)	Return terminal type code
GetTerminalFlags(*terminal_handle*)	Return terminal flags
GetTerminalLocation(*terminal_handle*, *array*)	Return terminal location
GetTerminalInstance(*terminal_handle*)	Return handle to associated subcell/device instance
IsTerminalFormal(*terminal_handle*)	Return 1 for formal terminals, 0 for subcell/device terminals
GetTerminalObject(*terminal_handle*)	Return handle to associated object
GetTerminalVgroup(*terminal_handle*)	Return virtual group number
GetTerminalLayer(*terminal_handle*)	Return associated layer name
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
ListGroupObjects(*group*)	Return list of objects in group
ListGroupDevContacts(*group*)	Return list of device contacts in group
ListGroupSubcContacts(*group*)	Return list of subcircuit contacts in group
ListGroupTerminals(*group*)	Return list of formal terminals in group
ListGroupTerminalNames(*group*)	Return list of formal terminal names in group
Physical Devices	
ListPhysDevs(*name*, *pref*, *index*, *l*, *b*, *r*, *t*)	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 name of physical subcircuit
GetPscIndex(*subckt_handle*)	Return index of physical subcircuit
GetPscDual(*subckt_handle*)	Return corresponding electrical subcircuit
GetPscBB(*subckt_handle*, *array*)	Return physical subcircuit bounding box
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
Electrical Nodes	
`EnableNodeMap(`*on*`)`	Enable/disable node name mapping
`GetNumberNodes(`*on*`)`	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
`ListNodeTerminals(`*node*`)`	Return list of connected terminals
`ListNodeTerminalNames(`*node*`)`	Return list of connected terminal names
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

D.1 Control Functions

D.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, or a library file. If 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 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.

The *symname* must be the name after any aliasing is applied.

When reading an archive file, the dimensions are scaled according to the current scale for reading. This scale can be set with the `SetConvertScale` function, or from the **Conversion** – **Import** panel in *Xic*.

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 symbols but the symbol 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*`)`
This function will read a cell (and subcells) into memory, but unlike the `Edit` function, 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.

When reading an archive file, the cell is scaled according to the current scale for reading. This scale can be set with the `SetConvertScale` function, or from the **Conversion − Import** panel in *Xic*.

The return value is 1 on success, 0 otherwise.

(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.

(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 subcels ls "`$$`", thus for example using "`! $$`" would delete all geometry but retain the subcells.

The return value is the number of objects deleted.

(int) `Push(`*object_handle*`)`
This function will push the editing context to the cell of the instance referenced by the handle. 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 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) `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) `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.

(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.

(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.

D.1.2 Database

(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.

`Clear(`*name*`)`
If *name* is not empty, any matching symbol and all its descendents are cleared from the database, unless they are referenced by another symbol not being cleared. If *name* is null or empty, the entire database is cleared. This function is obviously very dangerous.

`ClearAll()`
This will clear all symbols from the present symbol table, clear and delete any other symbol tables that may be defined, and reset the layer tables to their original state as defined in the technology file, deleting any layers created subsequently. This function does *not* automatically open a new symbol. 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. Only the top-level cells are searched, so filenames of subcells in *Xic* native hierarchies will not be found.

(int) `NumCellsInMem()`
This function returns an integer giving the number of cells in the current symbol table.

(stringlist_handle) `ListCellsInMem()`
This function returns a handle to a list of strings, sorted alphabetically, giving the names of all cells in the current symbol table.

(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.

D.1.3 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	Subwindow (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	Subwindow (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 subwindows. 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 subwindows. 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 subwindows 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 subwindows. The function returns 0 if the argument does not correspond to an existing window, 1 otherwise.

(int) `ShowSymbolic(`*show*`)`
This sets or unsets symbolic display mode of the electrical part of the current cell, according to the state of the boolean argument. The return value is always 1. The function has no effect if an active symbolic representation of the cell does not exist.

D.1.4 Exit

`Exit()`
Calling this function terminates execution of the script.

`Halt()`
Calling this function terminates execution of the script, equivalent to `Exit`.

D.1.5 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, and will disappear when a new current cell is chosen. 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 `"t"`
Arguments: *xl*, *xr*, *yl* [, *yu*, *attribute*]
Draw an open triangle. The two lower vertices are *xl*,*yl* and *xr*,*yl*. The third vertex is (*xl*+*xr*)/2,*yu*. If *yu* is not given, it is set to make the triangle equilateral.

Type: 4 or `"c"`
Arguments: *xc*, *yc*, *rad* [, *attribute*]
Draw a circle of radius *rad* centered at *xc*,*yc*.

Type: 5 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: 6 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.

D.1.6 Ghost Rendering

(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.

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) 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 **Constrain 45** button is active, 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).

D.1.7 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*`,` *radius*`,` *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 circle containing the arc. The remaining arguments are the radius, 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*`,` *transform*`)`
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 next argument is a transformation code, which allows the text to be rotated and/or reflected about the anchor point. The bits in this code have the following effects:

Bits	Effect
0–1	00-no rotation, 01-90, 10-180, 11-270
2	mirror y after rotation
3	mirror x after rotation and mirror y
4	shift rotation to 45, 135, 225, 315
5–6	horiz justification 00,11 left, 01 center, 10 right
7–8	vert justification 00,11 bottom, 01 center, 10 top

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.

D.1.8 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. Note: This is ignored if the rendering driver has "`NoAlt`" set, use `ShowAxes` in this case.

(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. Note: This is ignored if the rendering driver has "`NoAlt`" set, use `ShowGrid` in this case.

(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. Note: This is ignored if the rendering driver has "`NoAlt`" set, use `SetGrid` in this case.

(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. Note: This is ignored if the rendering driver has "`NoAlt`" set, use `SetGridStyle` in this case.

(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. Note: This is ignored if the rendering driver has "`NoAlt`" set, use `SetGridOnTop` in this case.

(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.

D.1.9 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.

D.1.10 Mode

`Mode(`*window*`, `*mode*`)`
This function switches *Xic* between physical and electrical modes, or switches subwindows 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 subwindows, in which case the viewing mode of that subwindow is set. The subwindow number is the same number as shown in the window title bar. The second argument is 0 for physical mode, nonzero for electrical mode. The return value is the new mode setting (0 or 1) or -1 if the indicated subwindow is not active.

(int) `CurMode(`*window*`)`
This function returns the current mode (physical or electrical) of the main window or subwindows. The argument is an integer 0–4 where 0 represents the main window (and the application mode) and 1–4 represent subwindow viewing modes. The return value is 0 for physical mode, 1 for electrical mode, or -1 if the indicated subwindow does not exist. This function is identical to `GetWindowMode`.

D.1.11 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.

D.1.12 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 submenu 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.

D.1.13 Technology File

`Techfile(`*techname*`)`
This function currently does nothing.

(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**.

D.1.14 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.

`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).

(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 1.2.4) variable, and the path is enclosed in parentheses. See the **!set** command description in 14.19 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.

D.1.15 *Xic* Version

(string) `VersionString()`
This function returns a string containing the current *Xic* version in a form like "`2.5.40`".

D.2 Utility Functions

D.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.

D.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.

D.2.3 Cell Name Enumeration

(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.

(stringlist_handle) `CellsHandle(`*cellname*`, `*depth*`)`
This function returns a handle to a list of subcell names found in *cellname*, to the given *depth*. If *cellname* is null or empty, the current cell is assumed. 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.

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).

(string) `GenCells(`*stringlist_handle*`)`
This function returns a string containing the name of one of the cells in the hierarchy of the current cell. The argument is the return value from `InitGen` or `CellsHandle`. A different cell is returned for each call. The null string is returned after all cells have been returned.

Example:

This script will list all of the cells in the current hierarchy:

```
i = InitGen()
while ((name = GenCells(i)) != 0)
   Print(name)
end
```

D.2.4 Error Reporting

The following functions provide an interface to the *Xic* error reporting and logging system.

(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.

The previous 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.

(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*`)`
Apply a new error message index to the string, and append the string the the error log file. The return value is the index of the new message, or 0 if the *string* is empty or null.

D.2.5 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 1 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.

D.2.6 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.

D.2.7 Miscellaneous

(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) `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 number
"`string`"	variable is a string
"`array`"	variable is an array
"`zoidlist`"	variable is a zoidlist
"`lexper`"	variable is a lexper
"`handle`"	variable is a handle to something

D.2.8 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`.

D.2.9 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 generated 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.

D.2.10 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.

(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.

D.2.11 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 ¡i¿string¡/i¿ 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 pointer to the file name part of a full path given in the string.

(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) `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.

D.2.12 String and Array Size

(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

D.3 Operating System and Input/Output Functions

D.3.1 Archive File Context

There is an internal data structure for saving information from a cell hierarchy in compact form. This is accessible from a set of script functions described below.

This allows very large data files to be scanned, and opened in skeleton mode. When opening a file in skeleton mode in the "normal" way, the entire hierarchy of cells and instances is saved, minus only the geometry. One could still run out of memory with large hierarchies, since all cells/instances are saved, whether used or not, and the main database is not efficient for storing the limited information really needed. The functions below get around this by pruning the skeleton to only the cells/instances actually needed to represent the data window before creating the in-memory hierarchy.

This capability applies to physical data only, and the following script functions expose the implementation. Note that there is a handle type to support this data structure.

(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 whose bits are flags that represent the type of information to return. See the description of the `CxInfo` function for information about these flags and the corresponding output.

This function is similar to the **!fileinfo** command.

(cx_handle) `GetArchiveContext(`*filename*`,` *scale*`,` *save_pcinfo*`)`
This function returns a handle to a context representation obtained from the archive file given as the argument. The file is opened from the library search path, if a full path is not provided. The context representation is a data structure that provides information about the hierarchy in compact form, and does not use that main database. The second argument is a scale factor that will be applied as the file is read, and must be between .001 and 1000. The third argument is a boolean that causes the context to store extra per-cell information. This information is used by

the `CxEstFlatMemoryUse` function, and the argument should be nonzero if this function is going to be used with the returned context. Otherwise, the third argument can be zero.

If a boolean true is passed for *save_pcinfo*, then the counts reported by `CxInfo` in the context represent only those objects on layers that pass any layer filtering from the LayerList/UseLayerList variables. Furthermore, in this case, the cell bounding boxes represent only objects on these layers, and subcells. In this mode, any read with the context should use the same layer filtering, or the bounding boxes of cells in memory won't match those computed in the context. The CxEstFlatMemoryUse function will report only the size potentially occupied by objects on these layers. If this argument is false, the counts and bounding boxes represent all layers, i.e., layer filtering is ignored.

The returned handle can be passed to other functions that require this data type. The `Close` function can be used to delete the context structure and free the handle.

The names of cells saved in the context reflect any aliasing that was in force at the time the context was created. The applicable aliasing is the same as if the file is read into memory with the **Conversion – Import** panel in the **Convert Menu**.

The random-access capability of cells through the context can be very slow if the file is compressed.

(int) `SaveArchiveContext(`*cx_handle*`, `*name*`)`
Store the context into a database, keyed by *name*. The *name* is an arbitrary short string, for example the file name. The context will remain in program memory until it is explicitly removed, or `ClearAll` is called. Contexts that are not saved are destroyed when the handle is closed, or the script terminates. A context can not be stored more than once, and the name must not already be in use for context storage. The function returns 1 if the context was successfully stored, 0 otherwise with possibly an error message retrievable with `GetError`.

(cx_handle) `RecallArchiveContext(`*name*`, remove)`
Recall a context from the database (as saved with `SaveArchiveContext`) and return a handle to it. If the integer *remove* is nonzero, it is removed from the database, otherwise not. A scalar 0 is returned if the context is not found or some other error occurs.

(int) `WriteArchiveContext(`*cx_handle*`, `*filename*`)`
This function will write a disk file representation of the archive context associated with the handle, into the file whose name is given as the second argument. Subsequently, the file can be read with `ReadArchiveContext` to recreate the context. The file has no other use and the format is not documented.

The archive context (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 context is potentially or actually in use.

The function returns 1 if the file was written successfully, 0 otherwise, with an error message likely available from `GetError`.

(int) `ReadArchiveContext(`*filename*`)`
This function returns a handle to an archive context created from the file whose name is passed as an argument. The file must have been created with `WriteArchiveContext`. On error, a scalar 0 is returned, with an error message probably available from `GetError`.

(string) `CxInfo(`*cx_handle*`, handle_or_filename, `*flags*`)`
This function provides information about the context indicated 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 whose bits are flags that represent the type of information to return. If this value is 0, all flags except for `INFO_FLAGS` will be taken as set. Otherwise the returned information will contain fields associated with the set bits:

Flag Name	Value	Description
`INFO_FILENAME`	`0x1`	File name.
`INFO_FILETYPE`	`0x2`	File type ("`CIF`", "`CGX`", "`GDSII`", or "`OASIS`").
`INFO_OBJCOUNTS`	`0x4`	Table of object counts.
`INFO_RECCOUNTS`	`0x8`	Table of record type counts (file format dependent).
`INFO_ESTSIZE`	`0x10`	Print estimated memory needed to read file into *Xic*.
`INFO_ESTCXSIZE`	`0x20`	Print size of data structure used to provide info.
`INFO_ESTSKSIZE`	`0x40`	Estimated memory required for skeleton.
`INFO_LAYERS`	`0x80`	List of layer names found, as for `CxLayers` function.
`INFO_TOPCELLS`	`0x100`	Top-level cells.
`INFO_ALLCELLS`	`0x200`	All cells.
`INFO_OFFSORT`	`0x400`	Sort cells by offset in archive file.
`INFO_OFFSET`	`0x800`	Print offsets of cell definitions in archive file.
`INFO_INSTANCES`	`0x1000`	List instances with cells.
`INFO_BBS`	`0x2000`	List bounding boxes with cells, and attributes with instances.
`INFO_FLAGS`	`0x4000`	List flags with cells and instances.
`INFO_DEPTHCNTS`	`0x8000`	Tabulate the number of cell instances at each hierarchy level.
`INFO_UNRESOLVED`	`0x10000`	List any cells that are referenced but not defined in the file.

The information provided by these flags is more fully described below.

`INFO_FILENAME`
: Print the name of the archive file for which the information applies.

`INFO_FILETYPE`
: Print a string giving the format of the archive file: one of "`CIF`", "`CGX`", "`GDSII`", or "`OASIS`".

`INFO_OBJCOUNTS`
: Print a table of object counts found in the archive file. The table contains the following keywords, each followed by a number.

Keyword	Descripption
`Records`	Total record count
`Symbols`	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

`INFO_RECCOUNTS`
: Print a table of the counts for record types found in the archive. This is format-dependent.

`INFO_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.

`INFO_ESTCXSIZE`
: *Xic* uses an "archive context" data structure to hold information after scanning a file. This

data structure has many uses. If this flag is set, an estimate of the size of this data structure is printed.

`INFO_ESTSKSIZE`
Print an estimate of the memory needed to read the archive in skeleton mode. This is the memory space used to hold the cell/instance hierarchy, without geometry.

`INFO_LAYERS`
Print a list of the layer names encountered in the archive, as for the `CxLayers` function.

`INFO_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 `INFO_ALLCELLS` is also given, only the names are listed, otherwise the cells are listed including the `INFO_OFFSET`, `INFO_INSTANCES`, `INFO_BBS`, and `INFO_FLAGS` fields if these flags are set. The list will be sorted as per `INFO_OFFSORT`.

`INFO_ALLCELLS`
All cells found in the file are listed by name, including the `INFO_OFFSET`, `INFO_INSTANCES`, `INFO_BBS`, and `INFO_FLAGS` fields if these flags are also given. The list will be sorted as per `INFO_OFFSORT`.

The following flags apply only if at least one of `INFO_TOPCELLS` or `INFO_ALLCELLS` is given.

`INFO_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.

`INFO_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.

`INFO_INSTANCES`
For each cell, the subcells used in the cell are listed. The subcell names are indented and listed below the cell name.

`INFO_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, $nx \times ny$ with spacings dx, dy.

`INFO_FLAGS`
For cells, the "area_ok" flag is shown. This will be set when the area configuration has been applied (with `CxSetArea`), and the cell is included within the hierarchy needed to represent the configured area. This will be zero in `FileInfo` and similar calls.

For instances, a similar flag is shown. If set, the instance is included in the configuration area. This will be zero in `FileInfo` and similar calls.

`INFO_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 given one count, the same as a single instance.

`INFO_UNRESOLVED`
This will list cells that are referenced but not defined in the file. These will also be listed if `INFO_ALLCELLS` is given. A valid archive file will not contain unresolved references.

This function is very similar to `FileInfo`.

(real) `CxEstFlatMemoryUse(`*cx_handle*`, cellname,` *array*`,` *counts_array*`)`
This function will return an estimate of the memory required to perform a `CxOpenFlat` call. The first argument must have been obtained from a `GetArchiveContext` call with the *save_pcinfo* argument set true (nonzero). The second argument is the name of the cell in the context to take as the top-level cell. If 0 is passed for this argument, the top-level cell of the context which appears first in the archive file will be used.

The third argument is an array of size four or larger that contains the rectangular area as passed to the `CxOpenFlat` 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 `CxOpenFlat` 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 `CxOpenFlat` call. This does not include normal overhead.

(int) `CxCompare(`*cx_handle1*`, cx_handle,` *cellname*`,` *layer_list*`,` *obj_types*`,` *skip_layers*`,` *geometric*`,` *array*`)`
This will compare the contents of the given cell, somewhat similar to the **!compare** command, but unlike the command, only one cell is compared.

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 all differences without limit. Properties are ignored.

The arguments are:

cx_handle1
Handle to an archive context.

cx_handle2
Handle to an archive context, or zero. If zero, the cell should exist in memory.

cellname
Name of cell to compare.

layer_list
String of space-separated layer names, or zero which implies all layers.

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.

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).

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 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 comparson, differences found.
2 The cell was not found in handle1.
3 The cell was not found in handle2.
4 The cell was not found in either handle.

(string) `CxFileName(`*cx_handle*`)`
This function returns a string containing the full path name of the file associated with the context handle argument. A null string is returned on error.

(string) `CxFileType(`*cx_handle*`)`
This function returns a string containing the file format of the archive file associated with the context handle passed. A null string is returned on error. Other possible returns are "`CIF`", "`GDSII`", "`CGX`", and "`OASIS`".

(stringlist_handle) `CxTopCells(`*cx_handle*`)`
This function returns a handle to a list of strings that contain the top-level cell names in the hierarchy associated with the context passed (physical cells only). The top-level cells are those not used as a subcell by another cell in the context. A scalar zero is returned on error.

(stringlist_handle) `CxLayers(`*cx_handle*`)`
This function returns a handle to a list of strings that contain the names of layers used in the context 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. Each unique combination or layer name is listed. A scalar zero is returned on error.

(string) `CxEstSkelSize(`*cx_handle*`)`
This return value is a string containing a floating-point number, which is an estimate in megabytes of the memory required to hold the context in a skeleton representation in the regular database. This takes into account the pruning implicit when `CxSetArea` is applied. A null string is returned on error.

(int) `CxCellBB(`*cx_handle*`,` *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 context. If *cellname* is passed 0, the first top-level cell found in the archive file will be used. 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. 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 `GetArchiveContext` is called, text labels will not contribute to the bounding box computation.

(int) `CxSetArea(`*cx_handle*`,` *cellname*`,` *l*`,` *b*`,` *r*`,` *t*`,` *perm*`)`
This will configure the context so that only cells and instances that overlap the rectangle given by the *l*,*b*,*r*,*t* arguments, when reflected to the coordinates of *cellname*, will be active. The top-level cell is taken as *cellname*. If *cellname* is passed 0, the first top-level cell found in the archive file will be used. Thus, the hierarchy if effectively "pruned", leaving only elements required to represent the given area in *cellname*.

The coordinates are given in microns. One is returned on success, zero otherwise.

(int) `CxClearArea(`*cx_handle*`)`
This will undo the configuration applied by `SetArea`, if the *perm* flag was not set. This function will have no effect otherwise. One is returned on success, zero if an error occurs.

(int) `CxSetSkipFlag(`*cx_handle*`,` *cellname*`,` *skip*`)`
This will set/unset the skip flag in the context for the cell named in *cellname* (physical only). If *cellname* is passed 0, the first top-level cell found in the archive file will be used. With the skip flag set, the cell is ignored in the context, i.e., the cell and its instances will not be included in output or when reading into memory. 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.

(int) `CxSetSkipEmpties(`*cx_handle*`,` *cellname* `[,` *left*`,` *bottom*`,` *right*`,` *top* `])`
This will set the skip flags of all physical cells in the hierarchy under *cellname* that contain no readable geometry or non-empty subcells. If *cellname* is passed 0, the first top-level cell found in the archive file will be used. If the optional box coordinates are given (in microns), the skip flags will be set for cells that have no geometry or subcells containing geometry that would appear in the given area in the top-level cell. This is useful when windowing is to be employed with clipping but not flattening, which can generate empty cells that otherwise wouldn't be caught.

This will examine the source file to determine cell content, and takes into account the current layer filtering. This can be called before writing (though `CxWrite` has its own flag for this purpose) or opening the hierarchy to remove empty cells. The return value is 1 on success, 0 otherwise.

(int) `CxClearSkipFlags(`*cx_handle*`)`
This will reset the skip flags for all cells in the context. The return value is 1 on success, 0 otherwise.

(int) `CxEdit(`*cx_handle*`,` *cellname*`)`
This will read the given cell and its descendents into memory and open the cell for editing, similar to the `Edit` function. The return value takes the same values as the `Edit` function return.

The *cellname*, if nonzero, must be the name of a cell in the context. This is the cell name after any aliasing that was in force when the context was created. If *cellname* is passed 0, the first top-level cell found in the archive file will be used.

If the context has been configured for area with `CxSetArea` for the cell, only the cells and instances needed to display the configuration area will be brought into memory. Cells will contain geometry that is outside of the configuration area, but not contain instances that are entirely outside of the configuration area.

(int) `CxSetupSkeleton(`*cx_handle*`,` *cellname*`,` *l*`,` *b*`,` *r*`,` *t*`)`
This function will create a skeleton hierarchy in memory from the context, using the area configuration if applied. If no area configuration was applied, the entire hierarchy under *cellname* is created. If *cellname* is passed 0, the first top-level cell found in the archive file will be used.

The cell created for *cellname* becomes the current cell, and the skeleton mode functions can be used in the normal way. The original archive file must remain available and not be modified, as cells are accessed through offsets into the file saved in the context when geometry is read.

The *l*, *b*, *r*, *t* parameters set the bounding box that will be applied when reading geometry (i.e., an implicit call to `SkelSetDataWin` with these parameters). These should be the same as or a sub-rectangle of the bounding box passed to `CxSetArea`, in microns, if an area configuration was applied. Otherwise, the rectangle should intersect with the bounding box of *cellname*.

The return value has the same interpretation as the return from the `Edit` command, with 1 indicating success and 0 failure but other values may appear in strange cases.

(int) `CxWrite(`*cx_handle*`,` *scale*`,` *cellname*`,` *array*`,` *clip*`,` *all*`,` *flatten*`,` *no_empties*`,` *outfile*`)`
This will write the cell named in the *cellname* string to the output file given in *outfile*. If *cellname* is passed 0, the first top-level cell found in the archive file will be used. 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.

If the context has been configured for area with `CxSetArea` for the cell, only the cells and instances needed to display the configuration area will be written.

The scale will multiply the scale factor provided to `GetArchiveContext` in output.

The *cellname*, which can not be null or empty, must be the name of a cell in the context. The is the cell name after any aliasing that was in force when the context was created.

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. This window should be equal to or contained in the window used to configure the context, if any.

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.

If the boolean value *no_empties* is nonzero, empty cells and their references will not appear in output.

The return value is 1 on success, 0 on error, or -1 if an interrupt was received.

(int) `CxOpenFlat(`*cx_handle*`,` *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. If a cell named *cellname* already exists in memory, it will be overwritten. If *cellname* is passed 0, the first top-level cell found in the archive file will be used.

If the context has been configured for area with `CxSetArea` for the cell, only the cells and instances needed to display the configuration area will be read.

The *scale* will multiply the scale factor provided to `GetArchiveContext`.

The *cellname*, which can not be null or empty, must be the name of a cell in the context. This is the cell name after any aliasing that was in force when the context was created.

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. This window should be equal to or contained in the window used to configure the context, if any.

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 `CxOpenFlat`, the memory use can be estimated by calling the `CxEstFlatMemoryUse` function.

The return value is 1 on success, 0 on error, or -1 if an interrupt was received.

D.3.2 Conversion

(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 **Conversion** pop-up.

which = 1
: Set the scale used when writing a file with the `To`*XXX* script functions or similar, or the **Conversion** – **Export** panel.

which = 2
: Set the scale used when reading a file into *Xic* with the `Edit` or `OpenCell` functions or similar, or from the **Conversion** – **Import** 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.

(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 nonzero will apply case conversion when writing an archive file from memory, and if zero indicates that case conversion will be applied when reading or format converting archive files.

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 flag.

(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 nonzero will apply case conversion when writing an archive file from memory, and if zero indicates that case conversion will be applied when reading or format converting archive files.

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 flag.

(int) `FromArchive(`*archive_file*`, `*destination*`)`
This function will read an archive (GDSII, CIF, CGX, or OASIS) file and translate the contents to another format. 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.

The scale factor used can be set with the `SetConvertScale` function. The windowing and flatting functions will be applied if the corresponding flags are set with the `SetConvertFlags` and `SetConvertArea` functions. The scale, flags and area rectangle can also be set from the **Conversion** pop-up in *Xic*.

The value 1 is returned on success, 0 otherwise, with possibly an error message available from `GetError`. This function is aliased to `FromCGX`, `FromCIF`, `FromGDS`, and `FromOASIS` for backward compatibility.

(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.

D.3.3 Conversion — Export

(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. The function returns 1 if the save was successful, 0 otherwise.

(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 **Conversion − Export** 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 **Conversion - Export** panel.

(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. The files are written according to the settings of the controls in the **Conversion − Export** panel, and/or the related variables. The scale factor used can be set with the `SetConvertScale` function.

(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 symbol name suffixed with "`.cgx`". The file is written according to the settings of the controls in the **Conversion − Export** panel, and/or the related variables. The scale factor used can be set with the `SetConvertScale` function.

(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 symbol name suffixed with "`.cif`". The file is written according to the settings of the controls in the **Conversion − Export** panel, and/or the related variables. The scale factor used can be set with the `SetConvertScale` function. The scale factor used can be set with the `SetConvertScale` function.

(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 symbol name suffixed with "`.gds`". The file is written according to the settings of the controls in the **Conversion − Export** panel, and/or the related variables. The scale factor used can be set with the `SetConvertScale` function.

(int) `ToGdsLibrary(`*gds_name*`,` *symbol_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 symbol given, but any cell is added once only. The resulting file will in general contain multiple top-level symbols. The file is written according to the settings of the controls in the **Conversion − Export** panel, and/or the related variables. The scale factor used can be set with the `SetConvertScale` function.

(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 symbol name suffixed with "`.oas`". The file is written according to the settings of the controls in the **Conversion − Export** panel, and/or the related variables. The scale factor used can be set with the `SetConvertScale` function.

(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 function returns 1 on success, 0 otherwise with an error message possibly available from `GetError`.

(int) `ToSpice(`*filename*`)`
This command 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, given a "`.bak`" extension. The return value is 1 on success, 0 otherwise.

(int) `WriteGeomSplit(`*fname*`,` *cname*`,` *bname*`,` *l*`,` *b*`,` *r*`,` *t*`,` *x0*`,` *y0*`,` *w*`,` *h*`,` *bw*`)`
This function will read the archive file *fname*, and split the geometry in *cname* into a collection of flat files covering a grid, in the manner of the `SkelWriteGeomSplit` function. If *cname* is 0, the top-level cell, if unique in the file, will be assumed. This function operates outside of skeleton mode, and should consume far less memory and may be faster than reading/writing in skeleton mode.

Each output file contains the geometry for a cell in a defined grid, including some possible overlap with neighboring grid cells. The *l*, *b*, *r*, *t* define a rectangle, in microns, which is the area over which the grid will be defined. This is typically the bounding box of the top-level cell. If all four entries are set to 0, the bounding box will be computed. The area can be smaller than the bounding box, which will truncate the grid, however objects are not clipped to this boundary.

The *x0*, and *y0* (in microns) set the grid origin relative to the origin of the top level cell. The *w* and *h* (in microns) set the periodicity of the grid, i.e., the unit cell width and height. The *bw*, in microns, is the amount by which the grid cell is bloated before geometry is extracted and written. This must be positive or zero.

The *bname* is a cell path name, with an extension which gives the type of file to create. One of the following extensions must be provided:

`.cgx`	CGX output
`.cif`	CIF output
`.gds`	GDSII output
`.oas`	OASIS output

The 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. The 0,0 grid cell has its lower left corner at *x0,y0* before bloating by the border width, and the indices increase with increasing x,y. If negative, the character '`m`' is used instead of a minus sign. Each file will consist of a single cell, whose name is tha name of the file with the extension stripped, containing the geometry for that grid cell.

In addition to the channel output files, an *Xic* cell file is written, using the basename without the extension, that references each of the channel files. Opening this cell will open all of the channel files, showing all geometry from the original file (memory limits permitting).

This function reads data directly from the source file to minimize memory use. All files are opened and written in parallel. The user may encounter system limitations on the number of file descriptors open simultaneously. The function returns 1 on success, 0 otherwise.

D.3.4 Conversion — Import

(int) `SetConvertFlags(`*use_window*`,` *clip*`,` *flatten*`,` *no_empties*`)`
This function sets the status of flags used in format conversions. The three arguments correspond to the **Use Window**, **Clip to Window**, **Flatten Hierarchy**, and **No Empties** buttons in the **Conversion** pop-up. A nonzero integer value will set the flag, 0 will reset the flag.

The flags will affect conversions initiated from this pop-up, and the `FromArchive` script function. 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 bits are:

flatten	`0x1`
use_window	`0x2`
clip	`0x4`
no_empties	`0x8`

(int) `SetConvertArea(`*l*`,` *b*`,` *r*`,` *t*`)`
This function sets the rectangular area used to filter or clip objects during format conversions. This rectangle can also be set from the **Conversion** pop-up. Use of the window can be enabled with the `SetConvertFlags` script function. The coordinates are in microns, in the coordinate system of the top level cell of the input file, after scaling (if any). The function always returns 1.

(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 **Conversion − Import** panel. If the integer argument is nonzero, the flag will be set. The return value is the previous state of the flag.

D.3.5 Skeleton Mode

(int) `SkelSetDataWin(`*l*`,` *b*`,` *r*`,` *t*`)`
This function sets the clipping box used when processing data that have been read in "skeleton" mode. The coordinates specify the box corners, in microns, in the coordinates of the current cell. Geometry that does not overlap this region with nonzero intersection area will be ignored, or not read by `SkelReadGeom`. The function returns 1 if the clipping box is set, 0 if the given box has no area. The clipping box is "cleared" by setting values to (+/-) `INFINITY`. The clipping box is automatically cleared if the skeletal current cell is cleared from memory with the `SkelClear` or

`Clear` functions, or `ClearAll` is called. The clipping only applies to cells read in skeleton mode (the SkeletonMode variable is set).

(int) `SkelClear(`*cellname*`,` *revert*`)`
This function clears skeleton hierarchies from memory. The *cellname* is the name of a top-level skeleton-mode cell, or a null string (or 0). If the name of a cell is given, that cell and its descendents are cleared. Otherwise, all skeleton-mode cells in memory are cleared. If the integer *revert* is 0, the cells will be cleared from memory. If nonzero, the cells will remain in memory, with their present content, but will behave in all respects like normal cells. It is not possible to reestablish skeleton mode for such cells. If the operation succeeds, 1 is returned, otherwise 0 is returned indicating an error. The `GetError` function can be used to obtain an error message in this case.

(int) `SkelClearGeom()`
This function clears the cache of geometry read in by the `SkelReadGeom` function from the current cell and its descendents. This applies only to hierarchies opened in "skeleton" mode (when the SkeletonMode variable is set). This frees the memory space used for the cached objects. This is implicitly called by `SkelReadGeom`, so it is not generally necessary to use this function. This function does not redraw the screen.

(int) `SkelReadGeom(`*depth*`)`
This function will first clear then read into memory all of the geometry for the current hierarchy, which is expected to have been opened in "skeleton" mode (i.e., with the SkeletonMode variable set). Only geometry which is within the current data window (as set with `SkelSetDataWin`) will be read. The argument is the depth in the hierarchy to read. This can be an integer, with 0 representing the current cell only, 1 will include its subcells, and so on. A negative value will read the entire hierarchy. This argument can also be a string starting with '`a`' to indicate reading of the entire hierarchy. This functions does not redraw the screen.

(int) `SkelWriteGeom(`*depth*`,` *filename*`,` *flat*`,` *clip*`)`
This function will write the cells and objects that have nonzero area overlap with the current skeleton window to a file. The current hierarchy must have been opened in skeleton mode (i.e., with the SkeletonMode variable set). Only geometry which is within the current data window will be written. The first argument is the depth in the hierarchy to write. This can be an integer, with 0 representing the current cell only, 1 will include its subcells, and so on. A negative value will read the entire hierarchy. This argument can also be a string starting with '`a`' to indicate reading of the entire hierarchy.

The type of file written is implied by the *filename*. If the *filename* is null or empty, native cell files will be created in the current directory. If the *filename* is the name of an existing directory, native cell files will be created in that directory. Otherwise, the extension of the file name 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.

If the third argument is 0, the existing cell hierarchy is maintained. If nonzero, the written hierarchy will be flattened, i.e., all objects will appear in the top-level cell. If the fourth argument is nonzero, objects written to the file will be clipped to the skeleton data window. The return value is 1 on success, 0 otherwise.

(int) `SkelWriteGeomSplit(`*basename*`,` *x0*`,` *y0*`,` *wid*`,` *hei*`,` *border*`)`
This function will split the hierarchy under the current cell, which must have been read in skeleton mode, and write the flattened geometry to a set of output files. This is similar to the `WriteGeomSplit` function. Each file contains the geometry for a cell in a defined grid, including some possible overlap with neighboring grid cells. The *x0*, and *y0* (in microns) set the grid origin relative to the origin of the current cell. The *wid* and *hei* (in microns) set the periodicity of the grid, i.e., the unit cell width and height. The *border*, in microns, is the amount by which the grid cell is bloated before geometry is extracted and written. This must be positive or zero.

If `SkelSetDataWin` has been called, only the objects in the file that overlap the specified data window will be included in the output. If no data window was specified, all objects in the file will be written.

The *basename* is a cell path name, with an extension which gives the type of file to create. One of the following extensions must be provided:

`.cgx`	CGX output
`.cif`	CIF output
`.gds`	GDSII output
`.oas`	OASIS output

The 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. The 0,0 grid cell has its lower left corner at *x0,y0* before bloating by the border width, and the indices increase with increasing x,y. If negative, the character '`m`' is used instead of a minus sign. Each file will consist of a single cell, whose name is tha name of the file with the extension stripped, containing the geometry for that grid cell.

In addition to the channel output files, an *Xic* cell file is written, using the basename without the extension, that references each of the channel files. Opening this cell will open all of the channel files, showing all geometry from the original file (memory limits permitting).

This function reads data directly from the source file to minimize memory use. All files are opened and written in parallel. The user may encounter system limitations on the number of file descriptors open simultaneously. The function returns 1 on success, 0 otherwise.

(int) `SkelWriteDigest(`*digest_file*`)`
This function writes a digest file for the skeleton hierarchy rooted at the current cell. A digest file is a simplified representation of the cell hierarchy, containing only the information needed to access the cell data in the archive file. In order to access the cells in the archive file in skeleton mode, one can open the archive file directly, or read the digest for the archive using `SkelReadDigest`. The latter method should be faster.

This function must be called while in skeleton mode (the SkeletonMode variable is set), with a skeleton hierarchy rooted at the current cell. The argument is the name of the digest file, and must be a valid file name. The function returns 1 on success, 0 otherwise.

(int) `SkelReadDigest(`*digest_file*`,` *archive_file*`)`
This function reads a digest file created previously with `SkelWriteDigest`, which applies to the archive file given in the second argument. If successful, upon return the current cell is set to the cell that was current when the digest was created, and all cells in the hierarchy should have their offsets and bounding boxes set. One can then use skeleton mode functions to read the area-filtered geometry, just as if the cell was opened with the **Edit** command or equivalent. This method should be much faster for large archives.

This command must be called in skeleton mode (the SkeletonMode variable is set) with no other skeleton hierarchy in memory. It is very important that the digest file match the archive file. If the

archive is modified in any way after the digest was produced, the file offsets in the digest will be wrong and errors or program crashes could result. There is at this point no consistency checking.

The function returns 1 if successful, 0 otherwise.

D.3.6 Assembly Stream

These functions implement a functionality similar to the **!assemble** command. Unlike **!assemble**, it is possible to set input conversion modes on the individual files.

(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 merging 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_context*`,` *scale*`,` *layer_list*`,` *use_layer_list*`,` *layer_aliases*`,` *cname_prefix*`,` *cname_suffix*`,` *ccvt*`)`
This function will add a source specification to a stream. The specification can refer to either an archive file, or to a context handle. 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.

stream_handle
Handle to the stream object.

file_or_context
This argument can be either a string giving a path to an archive file, or a context handle as returned from `GetArchiveContext`. If a context is passed, at least one cell instance must be specified with `StreamInstance`.

scale
A scale factor applied when writing. For a context, the scale will multiply the scale factor provided to `GetArchiveContext` in output.

layer_list
This can be a string containing a space-separated list of layer names. If 0 is given instead, the argument effectively contains the present value of the LayerList "**!set**" variable.

use_layer_list
If this value is a string starting with '`n`' or '`N`', objects on layers found in the *layer_list* will be skipped in output. If set to any other string, only objects found on layers in the *layer_list* will be written. If 0 is given, the argument is effectively the current value of the UseLayerList variable.

layer_aliases
: This can be set to a string containing "*name=alias*" layer name substitution assignments. Objects on layer *name* will be written on layer *alias*. If 0 is passed instead, and the UseLayerAlias variable is currently set, the argument is effectively the current contents of the Layer Alias table.

cname_prefix
: This can be set to a prefix string for cell name modification. If 0 is passed, this argument is effectively the current value of the InCellNamePrefix variable. Name modification is performed in the manner of that variable.

cname_suffix
: This can be set to a suffix string for cell name modification. If 0 is passed, this argument is effectively the current value of the InCellNameSuffix variable. Name modification is performed in the manner of that variable.

ccvt
: This integer controls case-conversion of cell names. If 1, upper case names are converted to lower case. If 2, lower case names are converted to upper case. Otherwise, no case conversion is done.

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*`,` *no_empties*`,` *flatten*`,` *array*`,` *clip*`)`

This function will add an instance name to the most recently added source file (using `StreamSource`). The *cellname* must match the name of a cell found in the archive specified in the source, including any aliasing in effect. There are two consequences: only the named cell and 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 scale factor for the cell.

At least one instance must be specified to a source supplied as a context description. A source must have been specified before this function can be called successfully.

The remaining arguments are ignored unless the extraction flag is set in the source. When extracting, a context structure is built for the source, which enables the operations to which these arguments pertain.

no_empties
: If the boolean value *no_empties* is nonzero, empty subcells will not be written to output.

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. This window should be equal to or contained in the window used to configure the context, if any.

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`.

D.3.7 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.

D.3.8 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) `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.

D.3.9 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(`*directory, filter*`)`
This function returns a handle to a list of names of files and/or directories in the given directory. If the *directory* 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.

D.3.10 Menu Buttons

(int) `SetButtonStatus(`*menu, button, set*`)`
This command sets the status of the specified button in the given menu, which must be a toggle button.

The first argument is a string giving the internal name of a menu. If the given name is null or empty, all of the menus in the main window will be searched. The internal menu names are as follows:

`file`	**File Menu**
`edit`	**Edit Menu**
`view`	**View Menu**
`trns`	**Transform Menu**
`attri`	**Attributes Menu**
`cnvrt`	**Convert Menu**
`drc`	**DRC Menu**
`extrc`	**Extract Menu**
`user`	**User Menu**
`help`	**Help Menu**
`main, side`	**Side Menu** (both equivalent)
`sub`*N*	**Viewport Menu**, $N = 1$–4

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 and only the first one or two characters have to match. Thus "Convert", "c", and "crazy" would all select the `cnvrt` menu, for example. One character is sufficient, except for 'e' (`extrc` 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 side menu and subwindows, the whole keyword must be supplied. The name "`side`" is equivalent to "`main`".

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 depressed. 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.

D.3.11 Mouse Input

(int) `Point(`*array*`)`
The argument is the name of a two component (or larger) array. This function blocks until a mouse 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, 1 for button 1, 2 for button 2, etc.

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).

D.3.12 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.

D.3.13 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.

D.3.14 Text Output

(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.

D.4 Geometry Creation and Management Functions

The following functions create or manipulate database objects.

D.4.1 General

`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.

D.4.2 Current Transform

(int) `SetTransform(`*angle*`,` *reflection*`,` *magnification*`)`
This function sets the "current transform" to the values provided. The floating point number *angle* is 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. This command is similar in action to the buttons in the **Current Transform** panel.

Examples:

Set rotation 180, mirror about Y axis:
```
SetTransform(180, 1, 1)
```
Set rotation 180, mirror about X axis:
```
SetTransform(180, 2, 1)
```
Set rotation 180, mirror about X,Y axes:
```
SetTransform(180, 3, 1)
```

(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.

(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
```

D.4.3 Grid

(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 subwindows. 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 subwindows. 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) `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 subwindows. The *interval* argument sets the grid line spacing, in microns. The *snap* number supplies the number of snap points per grid interval if positive, or the number of grid intervals per snap point if negative. Values from -10 – 10 are accepted for this parameter. Values of 0.01 or larger are valid for the *interval*, if not 0. Either argument can be 0, in which case the present parameter is retained. The function fails if the window argument does not correspond to an existing window.

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 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 subwindows. The function fails 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 subwindows. The snap number determines the number of snap coordinates between grid intervals if positive, or grid intervals per snap coordinate if negative. The function fails if the argument does not correspond to an existing window.

(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 subwindows. 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 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 subwindows. 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 subwindows. If the grid is shown below rendered objects, 0 is returned. If the window does not exist, -1 is returned.

(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 subwindows. The function fails if the window argument does not correspond to an existing window.

D.4.4 Layers

(int) `SetCurLayer(`*name*`)`
The string argument contains a layer name. Calling this function sets the current layer to the named layer. The function will fail if the named layer does not exist.

(int) `SetCurLayerAlias(`*longname*`)`
This function sets the long name of the current layer to the string given as an argument. The long name is an optional secondary name for a layer. Most if not all functions that take a layer name argument will also accept a long name. The long name can contain any characters, but it should not match the short or long name of another layer. Matching of the long name, as with the regular name, is case-sensitive. The function returns 1 if the long name is applied to the current layer, 0 if there is no current layer.

(int) `SetCurLayerDescr(`*description*`)`
This function sets the description of the current layer to the string given as an argument. The description is an optional text string associated with the layer. The function returns 1 if the description is applied to the current layer, 0 if there is no current layer.

(string) `SetCurLayerExKeyword(`*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 current layer, overriding existing settings and possibly causing incompatible or redundant existing keywords to be deleted. This is similar to the editing functions of the **Edit Extraction** command in the **Extract Menu**.

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
DarkField
Symbolic
Resistance
Capacitance
Tranline
Thickness
Rho
Sigma
EpsRel
Lambda
```

(int) `RemoveCurLayerExKeyword(`*keyword*`)`
This will remove the specification for the extract keyword given in the argument from the current layer. The argument must be one of the keywords listed for `SetCurLayerExKeyword`. The return value is 1 if a specification was removed, 0 otherwise.

(int) `NewCurLayer(`*name*`)`
The current layer is set to the named layer. If the named layer does not exist, it will be created. If the name is not a valid CIF layer name (four characters maximum, all alphanumeric) and a new layer is created, its name will be truncated to the first four characters, and any non-alphanumeric characters are replaced with 'X'. The original name will be saved as the long name (alias) for the new layer. The function will fail if the name is null, empty, or contains only white space.

(string) `GetCurLayer()`
This function returns a string containing the name of the current layer.

(string) `GetCurLayerAlias()`
This function returns a string containing the long name of the current layer. If no long name has been set, a null string is returned.

(string) `GetCurLayerDescr()`
This function returns a string containing the description of the current layer. If no description has been set, a null string is returned.

(int) `AddLayer(`*layer_name*`)`
This adds the named layer to the end of the layer table, much like "`!ltab add` *layer_name*" and "`!layer` *layer_name*". The string passed should contain only one name, unlike "`!ltab add`". This returns 1 on success, 0 on error. See the **!ltab** and **!layer** command descriptions.

(int) `RemoveLayer(`*layer_name*`)`
This removes *layer_name* from the layer table, much like "`!ltab remove` *layer_name*". The string passed should contain only one name, unlike "`!ltab remove`". This returns 1 on success, 0 on error.

(int) `RenameLayer(`*oldname*`,` *newname*`)`
This renames the layer named in *oldname* to *newname*, much like "`!ltab rename`". This returns 1 on success, 0 on error.

(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 (left to right) order. If the argument is nonzero, the list is in descending order. The layers used in the current 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 simply an alias for `ListNext`.

(int) `IsLayerConductor(`*lname*`)`
The argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The function returns 1 if the Conductor keyword is given or implied for the layer, 0 otherwise. This attribute applies to physical layers only.

(int) `IsLayerRouting(`*lname*`)`
The argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The function returns 1 if the Routing keyword is given for the layer, 0 otherwise. This attribute applies to physical layers only.

(int) `IsLayerGround(`*lname*`)`
The argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The function returns 1 if one of the GroundPlane keywords was given for the layer, 0 otherwise. This attribute applies to physical layers only.

(int) `IsLayerContact(`*lname*`)`
The argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The function return 1 if the Contact keyword is given for the layer, 0 otherwise. This attribute applies to physical layers only.

(int) `IsLayerVia(`*lname*`)`
The argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The function returns 1 if the Via keyword is given for the layer, 0 otherwise. This attribute applies to physical layers only.

(int) `IsLayerDarkField(`*lname*`)`
The argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The function returns 1 if the DarkField keyword is given or implied for the layer, 0 otherwise. This attribute applies to physical layers only.

(int) `IsLayerVisible(`*lname*`)`
The argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The function returns 1 if the layer is currently visible (i.e., has the visibility flag set), 0 otherwise.

(int) `SetLayerVisible(`*lname*`,` *visible*`)`
The first argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The second argument will set the layer visibility, visible if nonzero, invisible otherwise. The previous visibility status is returned. The display is not redrawn, use the `Redraw` function if necessary.

(real) `GetLayerMinDimension(`*lname*`)`
The argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The return value is the minimum dimension in microns for the layer. This is obtained from the applied design rules for physical layers, and the return value will be 0 for electrical layers.

(real) `GetLayerWireWidth(`*lname*`)`
The argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The function returns the default wire width for the layer.

(real) `GetLayerThickness(`*lname*`)`
The argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The function returns the value of the Thickness parameter given for the layer, if any. This attribute applies to physical layers only.

(real) `GetLayerRho(`*lname*`)`
The argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The function returns the resistivity of the layer as given by the Rho or Sigma parameters, if given. These attributes apply to physical layers only.

(real) `GetLayerEps(`*lname*`)`
The argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The function returns the relative dielectric constant for the layer, as given by the EpsRel parameter if applied. This attribute applies to physical layers only.

(real) `GetLayerLambda(`*lname*`)`
The argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The function returns the value of the Lambda parameter for the layer, if given. This attribute applies to physical layers only.

(real) `GetLayerResis(`*lname*`)`
The argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The function returns the value of the Resistance parameter for the layer, if given. This attribute applies to physical layers only.

(real) `GetLayerCap(`*lname*`)`
The argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The function returns the per-area capacitance for the layer, if given. This attribute applies to physical layers only.

(real) `GetLayerCapPerim(`*lname*`)`
The argument is a string giving the name of a layer in the current display mode. If the string is empty or null, the current layer is understood. The function will fail if a name is given which is not a layer name. The function returns the per-perimeter capacitance for the layer, if given. This attribute applies to physical layers only.

(int) `AddLayerGdsOutMap(`*lname*`,` *layer_num*`,` *datatype*`)`
This function will add a mapping from the layer named in the first argument (a 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 an error occurred, in which case no mapping has been created. If the layer name is not found in the layer table for the present mode (physical or electrical), or the layer number or data type number is out of range, 0 is returned. The acceptable range for the layer number and data type is [0 – 255].

(int) `RemoveLayerGdsOutMap(`*lname*`,` *layer_num*`,` *datatype*`)`
This function will remove a GDSII output layer mapping for the layer named in the first argument (a string). The mapping may have been applied in the technology file, with the **Conversion Parameter Editor**, panel, 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 – 255], 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 name can't be found in the layer table for the present mode (physical or electrical), or if the layer number or data type is out of range. Otherwise, the return value is the number of mappings removed.

(int) `AddLayerGdsInMap(`*lname*`,` *string*`)`
This function adds a GDSII input mapping record to the layer whose name is given as the first argument. 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 or other error, including the named layer not being found in the layer table for the current mode (physical or electrical). The function returns 1 if the mapping is successfully added.

(int) `ClearLayerGdsInMap(`*lname*`)`
This function deletes all of the GDSII input mappings applied to the layer named in the argument. These mappings may have been applied through the technology file, added with the **Conversion Parameter Editor**, 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(`*lname*`,` *datatype*`)`
This function assigns a data type to be used for objects with the DRC skip flag set. The first argument is the name of the (physical) layer. The second argument is the data type in the range [0 – 255], 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.

D.4.5 Layer Alias Table

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 variable and the script functions described below. There is also a **!lal** command which provides similar functionality.

(int) `ReadLayerAliases(`*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-byte hex number, using upper case, where the left two bytes represent the hex value of the layer number, zero padded, and the right two bytes represent the zero padded datatype number. The function returns 1 on success, 0 otherwise.

(int) `DumpLayerAliases(`*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) `ClearLayerAliases()`
This function will remove all entries in the layer alias table. The function always returns 1.

(int) `AddLayerAlias(`*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) `RemoveLayerAlias(`*lname*`)`
This function removes any alias for `lname` from the layer alias table. The function always returns 1.

(string) `LayerAlias(`*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.

D.4.6 Selections

(int) `SetLayerSpecific(`*state*`)`
This function will set layer-specific selection mode if the argument is nonzero, or normal mode otherwise. The return value is 1 or 0 representing the previous layer-specific 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) `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) `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.

(int) `Deselect()`
This function deselects all selected objects.

D.4.7 Object Management by Handles

The following functions provide a fairly complete interface to database objects. In general, objects in electrical views can be accessed for reading, but modification is prohibited.

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, except that the objects can not be modified. The functions that modify database objects will fail quietly (returning 0). The `ObjectCopy` function can be used to create a new database object from a copy. Lists of copies can be used in the first argument to the clipping functions such as `ClipTo`, but not the second. 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) `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) `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).

(object_handle) `BloatObjects(`*object_handle*`, `*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 third 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 fourth 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 (14.14.4) for documentation of the algorithms available.

The `ObjectDelete` function can be called to delete the old objects. The `ObjectCopy` 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*`,` *dimen*`,` *lname*`,` *mode*`)`
This function creates new polygon copies that cover the edges of the figures in the passed handle. The new polygons are like wires with width $2 * dimen$ (*dimen* is given in microns) centered on the figure edge. If a layer name is passed as a third argument, new polygons will be assigned to that layer, which will be created if it does not exist. If the argument is null or empty, new polygons will be assigned to the same layer as the original polygon. If the boolean argument *all* is nonzero, all of the objects in the passed list will be processed, otherwise only the object at the head of the list will be processed.

The `ObjectDelete` function can be called to delete the old objects. The `ObjectCopy` 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*`,` *dimen*`,` *lname*`)`
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 second argument is the minimum height or width in microns of rectangles created to approximate the non-Manhattan parts. The third 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 function will fail if the handle argument is not an object handle, or the size 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.

The `ObjectDelete` function can be called to delete the old objects. The `ObjectCopy` 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.

(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 `ObjectDelete` function can be called to delete the old objects. The `ObjectCopy` 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*`,` *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. 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 list passed as an argument. The third argument is an integer which if nonzero indicates a vertical decomposition, otherwise a horizontal decomposition is produced. The objects passed are untouched. The `ObjectDelete` function can be called to delete the old objects. The `ObjectCopy` 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) `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) `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.

(int) `ObjectDelete(`*object_handle*`)`
Calling this function will delete the object referenced by the handle from the current cell. The handle will point to the next object in its list. This function will fail if the handle passed is not a handle to an object list. If the object is deleted, 1 is returned, otherwise 0 is returned.

(int) `ObjectSelect(`*object_handle*`)`
This function will select the object referenced by the handle, if it is currently not selected, and the object is in the current cell. The function returns 1 if the object was previously not selected and was selected, otherwise 0 is returned. It is not possible to select object copies, 0 is returned if the passed handle represents copies. This function will fail if the handle passed is not a handle to an object list.

(int) `ObjectDeselect(`*object_handle*`)`
If the object pointed to by the handle is selected, calling this function will deselect the object. The function returns 1 if the object was previously selected and was deselected, otherwise 0 is returned. This function will fail if the handle passed is not a handle to an object list.

(int) `ObjectMove(`*object_handle*`,` *refx*`,` *refy*`,` *x*`,` *y*`,` *all*`)`
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) `ObjectMoveToLayer(`*object_handle*`,` *refx*`,` *refy*`,` *x*`,` *y*`,` *all*`,` *oldlayer*`,` *newlayer*`)`
This is similar to the `ObjectMove` function, but allows layer change. If *newlayer* is 0, null, or empty, *oldlayer* is ignored and the function behaves identically to `ObjectMove`. 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 `ObjectMove`, i.e., the layer arguments are ignored.

(int) `ObjectCopy(`*object_handle*`,` *refx*`,` *refy*`,` *x*`,` *y*`,` *all*`)`
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 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.

(int) `ObjectCopyToLayer(`*object_handle*`,` *refx*`,` *refy*`,` *x*`,` *y*`,` *all*`,` *oldlayer*`,` *newlayer*`)`
This is similar to the `ObjectCopy` function, but allows layer change. If *newlayer* is 0, null, or empty, *oldlayer* is ignored and the function behaves identically to `ObjectCopy`. 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 `ObjectCopy`, i.e., the layer arguments are ignored.

(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*`,` *touchok*`,` *useall*`,` *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 *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 *useall* 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 *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.

(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) `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 modify objects in physical views only. 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.

(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, or objects in electrical views. 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. It is unlikely that this information would be useful to the user. This function will fail if the handle passed is not a handle to an object list. A stale handle will return 0.

(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. This has relevance for physical boxes, polygons, and wires. The group number is set by the extraction system and is used for connectivity testing. This function will fail if the handle passed is not a handle to an object list. A stale handle will return 0.

(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. It applies in physical views only. 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. This applies to physical-mode labels only. 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) `GetLabelXform(`*object_handle*`)`
This function returns the orientation code of the label referenced by the handle, or 0 if the object is not a label. The orientation code is a bit field with the following significance:

bits	description
0–1	0-no rotation, 1-90, 2-180, 3-270
2	mirror y after rotation
3	mirror x after rotation and mirror y
4	shift rotation to 45, 135, 225, 315
5–6	horiz. justification 00,11 left, 01 center, 10 right
7–8	vert. justification 00,11 bottom, 01 center, 10 top
9–10	font

This function will fail if the handle passed is not a handle to an object list.

(int) `SetLabelXform(`*object_handle*`,` *xform*`)`
This function will apply the given orientation code, as defined for `GetLabelXform`, to the label referenced by the handle. If the operation is successful, 1 is returned, otherwise 0 is returned. Only physical-mode labels can be changed. This function will fail if the handle passed is not a handle to an object list.

(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) `GetInstanceName(`*object_handle*`)`
This function returns the 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.

(int) `SetInstanceName(`*object_handle*`,` *newname*`)`
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.

D.4.8 Pseudo-Flat Generator Functions

(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 D.4.7).

(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.

D.4.9 Clipping Functions

(int) `ClipAround(`*object_handle1*`,` *object_handle2* `[,` *all1*`,` *all2*`])`
This function will clip out the pieces of objects in the second handle list that intersect with objects in the first handle list. The third and fourth arguments are optional, and default to 0 if not given. If *all1* is nonzero, all objects in the first handle are used for clipping, otherwise only the first object is used. If *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*`,` *object_handle2* `[,` *lname*`,` *all1*`,` *all2*`])`
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. The remaining arguments are optional, and default to 0 if not given. The *all1* and *all2* are flags specifying whether to use all objects in the respective lists, or just the first, as in `ClipAround`. Unlike `ClipAround`, the second handle can reference object copies.

If *lname* is given and 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) `ClipAroundAll(`*object_handle1*`,` *object_handle2*`)`
This function is deprecated — see `ClipAround`.
This function is similar to the `ClipAround` function, however all items in both object lists are processed. New objects created are added to the front of the *object_handle2* list. Objects that are deleted are removed from both lists. Objects that appear in both lists will be deleted, unlike `ClipAround`. `HandlePurgeList` can be called to eliminate these objects should this be a potential problem. The returned value is the total number of new objects created, or -1 if there was an error. Any object can appear in the *object_handle1* list, the bounding box is used for labels and subcells. Only boxes, polygons, and wires in the *object_handle2* list will be processed. The function fails it either handle does not reference a list of objects.

(object_handle) `ClipAroundAllCopy(`*object_handle1*`,` *object_handle2*`)`
This function is deprecated — see `ClipAround`.
This function is similar to `ClipAroundAll`, however no new objects are created in the database, and neither of the lists passed as arguments is touched. 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 objects referenced by the second handle that do not intersect the objects referenced by the first handle. The returned list can be used just as any list of object copies, however they are not copies of "real" objects. If no new object would be created by clipping, the function returns 0. Unlike `ClipAroundAll`, the second handle can reference object copies.

(int) `ClipTo(`*object_handle1*`,` *object_handle2* `[,` *all1*`,` *all2*`])`
This function will clip objects referenced by the second handle to the boundaries of objects referenced by the first handle. The third and fourth arguments are optional, and default to 0 if not given. If *all1* is nonzero, all objects in the first handle are used for clipping, otherwise only the first object is used. If *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*`,` *object_handle2* `[,` *lname*`,` *all1*`,` *all2*`])`
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. The remaining arguments are optional, and default to 0 if not given. The *all1* and *all2* are flags specifying whether to use all objects in the respective lists, or just the first, as in `ClipTo`. Unlike `ClipTo`, the second handle can reference object copies.

If *lname* is given and 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) `ClipToAll(`*object_handle1*`,` *object_handle2*`)`
This function is deprecated — see `ClipTo`.
This is similar to the `ClipTo` function, however all objects in the handle lists are processed. Only the parts of the boxes, polygons, and wires in the second object list that overlap one of the objects in the first list will remain. Labels and subcells in the second list are ignored. The newly created objects are added to the front of the second handle list. The return value is the number of new objects created, or -1 if either handle is empty or some other error occurred. The function will fail if either handle does not reference an object list.

(object_handle) `ClipToAllCopy(`*object_handle1*`,` *object_handle2*`)`
This function is deprecated — see `ClipTo`.
This function is similar to `ClipToAll`, however no new objects are created in the database, and neither of the lists passed as arguments is touched. 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 objects referenced by the second handle that intersect the objects referenced by the first handle. The returned list can be used just as any list of object copies, however they are not copies of "real" objects. If no new object would be created by clipping, the function returns 0. Unlike `ClipToAll`, the second handle can reference object copies.

(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*`,` *object_handle2*`,` *lname*`,` *all1*`,` *all2*`)`
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.

D.4.10 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 **Edit Menu**.

(int) `Bloat(`*dimen*`,` *mode*`)`
Each selected object is bloated by the given dimension, similar to the **!bloat** command. In layer-specific mode, only selected objects on the current layer are affected, otherwise all selected objects are affected. 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 (14.14.4) for documentation of the algorithms available.

(int) `Manhattanize(`*dimen*`)`
Each selected non-Manhattan polygon or wire is converted to a Manhattan polygon or box approximation, similar to the **!manh** command. In layer-specific mode, only selected objects on the current layer are affected, otherwise all selected non-Manhattan objects are affected. The argument is a size in microns representing the smallest dimension of the boxes created to approximate the non-Manhattan parts. 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 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.

(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.

(int) `Arc(`*x*`,` *y*`,` *rad1X*`,` *rad1Y*`,` *rad2X*`,` *rad2Y*`,` *ang_start*`,` *ang_end*`)`
This produces a circular or elliptical figure, providing functionality similar to the **round**, **donut**, and **arc** buttons in the side menu.

x, y	center coordinates
rad1X, rad1Y	x and y inner or outer radii
rad2X, rad2Y	x and y outer or inner radii
ang_start	starting angle in degrees
ang_end	ending angle 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 (flash) 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.

(int) `Sides(`*numsides*`)`
This sets the number of segments to use in generating round objects. The function returns the present value for this parameter. The value is reset if the argument is in the range 8 through 150. This is similar to the **sides** menu button in the side menu.

(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.

(int) `Label(`*text*`,` *x*`,` *y* `[,` *width*`,` *height*`,` *xform*`)]`
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 reference point, which is dependent on the current justification, as set with the `Justify` function or the optional *xform* argument. The default is the lower-left corner of the bounding box.

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 *xform* argument is an integer whose bits set transformation and justification attributes, and if given the `Justify` function and `UseTransform` function settings will be ignored, and these attributes will be set from *xform*. If *xform* is not given, the functions will set the justification and transformation. The bits in *xform* have the following significance:

Bits	Effect
0–1	00-no rotation, 01-90, 10-180, 11-270
2	mirror y after rotation
3	mirror x after rotation and mirror y
4	shift rotation to 45, 135, 225, 315
5–6	horiz justification 00,11 left, 01 center, 10 right
7–8	vert justification 00,11 bottom, 01 center, 10 top

This function always returns 1.

(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
LogoManhattan
LogoPretty
LogoPrettyFont
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) `Place(`*cell*`, `*x*`, `*y*`)`
This function places a subcell at the indicated location. The first argument is of string type and contains the name of the cell to place. It must be available on the present path, or already exist in memory. The second two arguments define the location. The reference is dependent upon the current setting of the selection in the place panel, and is either the cell's lower left corner or the cell's origin in physical mode, and is the location of the cell's reference terminal in electrical mode. The cell is transformed according to the current transform setting. The returned value is 1 on success, 0 otherwise.

(int) `PlaceTemplateArgs(`*cell*`, `*args*`)`
This sets the default template parameters used when instantiating the template cell named in *cell*. The second argument is a string giving the comma-separated *name=value* pairs that will be applied, when the cell is instantiated with the `Place` function or otherwise. Note that in graphical mode, the given string will be the default string when the user is prompted to enter the parameters. This function should be used when placing template cells in batch or server mode, and applies only to subsequent `Place` calls. Additional calls to `Place` will use these parameters for *cell*, until changed with another call to this function. In graphical mode, if the user modifies the string, the modified string will be used in subsequent placements.

This function actually manages an internal table of cellname/argument list associations. If 0 is given for both arguments, the table will be cleared. If the second argument is 0, the entry for the first argument, if any, will be removed from the table. The table is cleared when the script terminates.

The return value is always 1.

(int) `Replace(`*cellname*`, `*add_xform*`)`
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.

(int) `PlaceParams(`*llref*`, `*nx*`, `*ny*`, `*dx*`, `*dy*`)`
This function sets the parameters used for instance placement. If *llref* is nonzero, the lower-left corner of the bounding box will be placed at the reference, otherwise the cell's origin will be placed at the reference, in physical mode. This is ignored in electrical mode by cells with connections assigned, as the first connection is the reference in that case. The remaining parameters define an array for use in physical mode. For the unarrayed case, *nx* and *ny* should be one. The *dx* and *dy* are the spacing between cells in the array, and are ignored if the corresponding *nx*/*ny* is one. Array parameters are ignored in electrical mode.

(int) `Delete()`
This function deletes all selected objects from the database.

(int) `DeleteEmpties(`*recurse*`)`
This function will delete empty symbols 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 symbols are deleted. If the argument is nonzero, additional passes are done to delete

symbols that are newly empty due to their subcells being deleted on the previous pass. The top-level symbol is never deleted. The return value is the number of symbols deleted. No symbols will be deleted if the top-level symbol was opened in skeleton mode, and skeleton mode is active (the `SkeletonMode` variable is set).

(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. If in layer-specific mode, only the current layer will be erased, otherwise all layers will be erased. 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*`)`
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. This provides functionality similar to the **Copy** button in the **Edit 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*`)`
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 **Edit 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*`)`
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. The return value is 1 if there were no errors and something was rotated, 0 otherwise.

(int) `RotateToLayer(`*x*`, `*y*`, `*ang*`, `*oldlayer*`, `*newlayer*`)`
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.

(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*`, `*fast*`)`
The selected subcells are flattened into the current cell, recursively to the given depth. 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. If *fast* is nonzero, "fast mode" is used, i.e., there will be no undo capability. This provides functionality similar to the **Flatten** button in the **Edit Menu**.

(int) `CreateSymbol(`*symname*`, [`*orig_x*`, `*orig_y*`])`
This will create a new symbol 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 symbol of the same name already exists in memory. However, if the CrsymOverwrite variable is set, existing cells will be overwritten with the new data, and the function will succeed.

`Layer(`*string*`, `*depth*`, `*flat*`, `*undoable*`)`
This is very similar to the **!layer** command. The string is of the form

[`join|split|splitv`] *new_layer_name* [=] *layer_expression*

The first token of the string can be "`split`", "`splitv`", or "`join`" to indicate whether the created geometry is in the form of a set of trapezoids or a minimal number of complex polygons. The "`split`" decomposition favors horizontal orientation, whereas "`splitv`" favors vertical orientation

of the trapezoids. If this optional keyword is missing, the default is to use the joined form. The join operation is sensitive to the JoinMax... variables. The equal sign after the new layer name is optional.

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.

The *flat* argument is a flag (0 or not 0). If *flat* is nonzero, the layer is created in the current cell using geometry to *depth*. Otherwise, the layer is created in each subcell to *depth*, using geometry only from the cell.

The *undoable* argument is also a flag, and if nonzero the operation will be undoable with the **Undo** command or function. If zero, the operation will not be undoable, but execution may be faster and memory use lower.

There is no return value; the function either succeeds or will terminate the script on error.

D.4.11 Property Management by Handles

The following functions provide an interface for working with properties.

(prpty_handle) `PrptyHandle(`*object_handle*`)`
This function returns a handle to the list of properties of the object referenced by the passed object handle. If the passed value is 0, the returned handle will reference the list of properties of the current cell, either the electrical or physical part depending on the display mode.

(int) `PrptyNumber(`*prpty_handle*`)`
This function returns the number of the property referenced by the handle.

(string) `PrptyString(`*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.

(int) `PrptyNext(`*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) `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. If the handle is 0, the property will be created in the the current cell. The object must be defined in the current cell.

In physical mode, the *number* argument is an integer, within the acceptable range.

Physical cell	0 – 6999, 7105 (Flags), 7198 (Template Params), 7199 (Template Script), 7300 or larger.
Physical object	0 – 6999, 7198 (Template Params), 7200 or larger.

Adding a Flags, Template Params or Template Script property will automatically remove the old property, if any. Note that pseudo-properties can be applied to physical objects.

In electrical mode, it is possible to set the name, model, value, param, other and nophys properties if the handle references an electrical device, or the same properties except for model and value if the handle references an electrical subcell, or the param and other properties in the cell. In this case, the number argument is a *string*, and should be a prefix of "name", "model", "value", "param", "other", or "nophys". The single character string "n" implies name, and (additionally) "y" implies nophys. If the string is not recognized, other is assumed. 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) PrptyRemove(*object_handle*, *number*, *string*)
This function will remove properties matching the given *number* and *string* from the object referenced by the handle. If the handle value is 0, properties will be removed from the current cell.

In physical mode, the *number* argument is an integer, within the acceptable range. 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.

Physical cell	0 – 6999, 7105 (Flags), 7300 or larger.
Physical object	0 – 6999, 7300 or larger.

Note that template properties can not be removed.

In electrica mode, the name, model, value, param, other, and nophys properties can be removed from a device, and the same properties except for model and value can be removed from an electrical subcell, and param and other properties can be removed from the cell. In this case, the number argument is a *string*, which should be a prefix of "name", "model", "value", "param", "other", or "nophys". The single character string "n" implies name, and (additionally) "y" implies nophys. If the string is not recognized, other is assumed. 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.

(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 is a prefix of "name", "model", "value", "param", "other", or "nophys", which is appropriate for electrical devices. The single character string "n" implies name, and (additionally) "y" implies nophys. If the string is not recognized, other is assumed.

(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 is a prefix of "name", "model", "value", "param" "other", or "nophys". The single character string "n" implies name, and (additionally) "y" implies nophys. If the string is not recognized, other is assumed.

(string) GetProperty(*prpty_handle*, *number*)
This 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 is a prefix of "name", "model", "value", "param" "other", or "nophys", which is appropriate for electrical devices. The single character string "n" implies name, and (additionally) "y" implies nophys. If the string is not recognized, other is assumed. 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 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(*object_handle*, *number*)
This function returns the strings from properties or pseudo-properties for the object referenced by the handle, or from the current cell if the handle is 0. The function will fail if a nonzero handle argument does not reference a list of objects.

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. Note that pseudo-properties are currently defined for objects, not cells. 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 is a string, in which case it should be a prefix of one of "name", "model", "value", "param", "other", or "nophys". The single character string "n" implies name, and (additionally) "y" implies nophys. If the string is not recognized, other is assumed. This input is appropriate for obtaining property values from electrical devices.

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.

D.4.12 Other Property Management Functions

(int) AddProperty(*number*, *string*)
This function adds the property *number* and *string* to all selected objects. In physical mode, the *number* argument is an integer, within the acceptable range.

Physical object 0 – 6999, 7198 (Template Params), 7200 or larger.

Adding a Template Params property will automatically remove the old property, if any. Note that pseudo-properties can be applied to physical objects.

In electrical mode, it is possible to set the name, model, value, param, other, and nophys properties on electrical devices, or the same properties except for model and value on electrical subcells. In this case, the number argument is a *string*, and should be a prefix of "name", "model", "value", "param", "other", or "nophys". The single character string "n" implies name, and (additionally) "y" implies nophys. If the string is not recognized, other is assumed. 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 *number* argument is an integer, within the acceptable range.

Physical cell 0 – 6999, 7105 (Flags), 7198 (Template Params), 7199 (Template Script), 7300 or larger.

Adding a Flags, Template Params or Template Script property will automatically remove the old property, if any.

In electrical mode, it is possible to set the param and other properties in the cell. In this case, the number argument is a *string*, and should be a prefix of "`param`" or "`other`". If the string is not recognized, other is assumed.

The function returns 1 if the operation was successful, 0 otherwise.

(int) `RemoveProperty(`*number*`,` *string*`)`
This function will remove properties from selected objects. In physical mode, the *number* argument is an integer, within the acceptable range. The *string* argument can be passed 0, in which case this function will remove all properties with the number given from all selected objects. If a string is given, then only those properties for which *string* is a prefix of the value string will be removed.

Physical object 0 – 6999, 7300 or larger.

Note that template properties can not be removed.

In electrical mode, the name, model, value, param, other, and nophys properties can be removed from a device, and the same properties except for model and value can be removed from an electrical subcell. In this case, the number argument is a *string*, which should be a prefix to "`name`", "`model`", "`value`", "`param`", "`other`", or "`nophys`". The single character string "`n`" implies name, and (additionally) "`y`" implies nophys. If the string is not recognized, other is assumed. 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 *number* argument is an integer, within the acceptable range. The *string* argument can be passed 0, in which case this function will remove all properties with the given integer *number* from the physical part of the current cell. If a string is given, only those properties for which the string is a prefix to the value will be deleted.

Physical cell 0 – 6999, 7105 (Flags), 7300 or larger.

Note that template properties can not be removed.

In electrical mode, the param and other properties can be removed from the cell. In this case, the *number* argument is a *string*, which should be a prefix to "`param`" or "`other`". If the string is not recognized, other is assumed. 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.

D.4.13 Design Rule Checking

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*`)`
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*`)`
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) `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(`*left*`,` *bottom*`,` *right*`,` *top*`,` *file_handle*`)`
This function performs DRC over the rectangular area designated by the first four arguments. The *file_handle* is a value returned from the `Open` function, or 0. If a file handle is passed, output is recorded in the file. If 0 is passed, output goes to the error logger which appears on-screen. The number of errors detected is returned.

(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.

D.4.14 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.

D.4.15 Zoidlists 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
lexper	evaluate layer expression, use result

(int) `SetZref(`*thing*`)`
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 variable. This can be a zoidlist, or an object handle, or 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 never fails.

(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) `AdvanceZref()`
This function allows iteration over the area of a large cell by establishing a grid over the current cell area and incrementally setting the reference area (see `SetZref`) to elements of the grid. The grid is aligned from the lower-left corner and iteration advances right and up. The reference area is set to the intersection of the grid element area and the bounding box of the current cell. The size of the square grid elements is given by the LayerPartSize variable, or defaults to 100 microns if this variable is not set.

With the boolean 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 current cell bounding box, in which case 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 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(`*layername*`,` *depth*`)`
This function returns a zoidlist from the layer given in the first argument, which has the form *layername*[.*cellname*]. If the *cellname* extension is not given, the current cell is assumed. 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.

(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.

(zoidlist) `TransformZ(`*zoid_list*`,` *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*`,` *zoid_list*`,` *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 (14.14.4) for documentation of the algorithms available.

(zoidlist) `EdgesZ(`*dimen*`,` *zoid_list*`,` *mode*`)`
This returns a list of zoids that in some way describe edges in the zoid list passed. There are currently three modes, 0—2 as set by the third argument. The *dimen* is given in microns.

mode 0
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 1
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 2
For each zoid in the *zoid_list* 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*`,` *zoid_list*`)`
This function returns a new zoidlist which is a Manhattan approximation of the zoidlist passed as an argument (similar to the **!manh** command). The argument is the minimum rectangle width or height in microns used to approximate non-Manhattan pieces. All of the returned trapezoids are rectangles. The function will fail if the argument is smaller than 0.01.

(zoidlist) `RepartitionZ(`*zoid_list*`)`
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.

(lexper) `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*`,` *zoid_list*`,` *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) `TestCoverage(`*layer_expr*`,` *zoid_list*`,` *testfull*`)`
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 `ParseLayerExpr`. If the second argument is 0, the current reference zoidlist as set with `SetZref` is assumed. If the *testfull* argument is 0, the return values are 0 if there is no dark area, and 1 if dark areas exist. If *testfull* is nonzero, an additional return value of 2 is returned if the *zoid_list* is completely covered by the layer expression result (i.e., completely dark). This latter test if somewhat more expensive. The function will fail on a parse or evaluation error.

(object_handle) `ZtoObjects(`*zoid_list*`,` *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*`,` *zoid_list*`,` *join*`)`
This function creates a temporary layer using *longname*, and adds the content of the *zoid_list* 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*`)`
Save the zoidlist in a file, whose name is given in the first argument. The zoidlist can be recovered with `ZfromFile`. 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 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.

(zoidlist) `CxGetZlist(`*cx_handle*`,` *cellname*`,` *scale*`,` *array*`,` *clip*`,` *all*`)`
This function will create and return a trapezoid list created from objects read from the file associated with the passed context.

The UseLayerList and LayerList variables should be set to specify the layer or layers that will be processed. By default, all layers will contribute to the trapezoid list, which will in effect represent the union of all geometry.

If the context has been configured for area with `CxSetArea` for the cell, only the cells and instances needed to display the configuration area will be processed.

The *cellname*, which can not be null or empty, must be the name of a cell in the context. The is the cell name after any aliasing that was in force when the context was created.

The *scale* will multiply the scale factor provided to `GetArchiveContext` in the output trapezoids.

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. This window should be equal to or contained in the window used to configure the context, if any.

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.

(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.

D.4.16 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) `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.

(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 the table is not found or x,y is out of range, 0 is returned.

(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 the table can't be found, x,y is out of range, or some other error occurs.

D.4.17 Polymorphic Flat Database

There functions are related to creating and using "special" databases. This is a new feature that is under construction. A special database is a spatially sorted container for objects or trapezoids (not cell instances or cells), with varying internal formats. The following new script functions expose this functionality.

(int) `CxOpenOdb(`*cx_handle*`, `*scale*`, `*cellname*`, `*array*`, `*clip*`, `*dbname*`)`
This function will create a "special database" of the objects read through the archive context handle. The coordinates will be scaled by *scale* as read. The *cellname* is the name of the top-level cell whose hierarchy is to be read from the context. If this argument is zero, the first top-level cell found in the corresponding archive file will be understood. 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.

(int) `CxOpenZdb(`*cx_handle*`,` *scale*`,` *cellname*`,` *array*`,` *clip*`,` *dbname*`)`
This function will create a "special database" of the trapezoid representations of objects read through the archive context handle. The coordinates will be scaled by *scale* as read. The *cellname* is the name of the top-level cell whose hierarchy is to be read from the context. If this argument is zero, the first top-level cell found in the corresponding archive file will be understood. 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.

(int) `CxOpenZbdb(`*cx_handle*`,` *scale*`,` *cellname*`,` *array*`,` *clip*`,` *dbname*`,` *dx*`,` *dy*`,` *bx*`,` *by*`)`
This will open a database similar to `CxOpenZdb`, however the trapezoids will be saved in binned lists. Unlike `CxOpenZdb`, the array can not be passed 0, a rectangular region must be specified. The *dx*, *dy* are the grid spacing values for the bins, in microns. These values must be positive. The *bx*, *by* are 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.

(int) `GetObjectsOdb(`*dbname*`,` *layer_list*`,` *array*`)`
This returns a handle to a list of objects, extracted from a named database created with `CxOpenOdb`. The first argument is a database name string as given to `CxOpenOdb` (This function will not work with `CxOpenZdb`). 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 `CxOpenOdb`, 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.

(int) `GetZlistDb(`*dbname*`,` *layer_list*`,` *zoidlist*`)`
This returns a zoidlist associated with a layer, extracted from a named database created with `CxOpenOdb`, `CxOpenZdb`, or `CxOpenZbdb`. The first argument is a database name string as given to `CxOpenOdb` or equivalent. The second argument is the associated layer name. The third argument is the reference trapezoid list. If the database was opened with `CxOpenOdb` or `CxOpenZdb`, the returned zoidlist will be clipped to the reference list. If the database was opened with `CxOpenZbdb`, the trapezoids for the bin containg 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.

(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 `CxOpenOdb` 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.

(int) `DestroyDb(`*dbname*`)`
This function will free and clear the special database named in the argument. This is the database

name as given to `CxOpenOdb` or equivalent. If the argument is 0, then all special databases will be freed and cleared. This function always returns 1.

D.5 Extraction Functions

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.

D.5.1 Menu Commands

(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**	NoPnet
`d`	**devs**	NoPnetDevs
`s`	**spice**	NoPnetSpice
`b`	**list bottom-up**	PnetBottomUp
`g`	**show geometry**	PnetShowGeometry
`c`	**include wire cap**	PnetIncludeWireCap
`a`	**list all cells**	PnetListAll

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**	NoEnet
`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

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 the the **Show Paths** and **Quick Paths** commands.

The return value is a handle to a list of object copies, or 0 if no objects are found.

(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 `Resistance` layer keyword gives the ohms-per-square of the material. If not set, a value of 1.0 is assumed.

For Inductance:
: The `Tranline` 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(`*x*`,` *y*`,` *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 algorthm used by the `ExtractRL` function. The resistance of the connected network is then computed, with respect to the terminals specified.

The first two arguments are a coordinate point that is expected to fall on a wire net (similar to the `FindPath` function). The net is recognized using the **Quick Paths** algorithm. Invisible layers are not included.

The third 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 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 return value is always 1. The function will fail (terminating the script) if an error is encountered.

D.5.2 Terminals

(int) `ModifyTerminal(`*xe*`,` *ye*`,` *xp*`,` *yp*`,` *name*`,` *lname*`,` *type*`,` *remove*`)`
This is a rather complicated function used to add, modify, or remove formal terminals of the current cell. The arguments are as follows:

xe, *ye* (real)
The coordinates, in microns of the terminal in the electrical schematic.

xp, *yp* (real)
The coordinates, in microns, of the terminal in the physical layout.

name (string)
The terminal's name.

lname (string)
Physical layer name associated with the terminal.

type (string)
The terminal type, one of `INPUT`, `OUTPUT`, `INOUT`, `SUPPLY`, `GROUND`.

remove (integer)
Non-zero will remove terminal.

If a terminal already exists which matches the given name, or either the physical or electrical coordinates, the existing terminal will be updated, or removed if *remove* is nonzero. Otherwise, if *remove* is zero, a new terminal will be added with the given characteristics. The *name*, *lname*, and *type* arguments can be passed 0, in which case a default will be used.

If the electrical coordinates do not match a "node" where a connection can occur, the terminal will be "virtual".

The function returns 1 if the operation succeeded, 0 otherwise.

(string) `GetTerminalName(`*terminal_handle*`)`
This will return a string containing the name of the terminal referenced by the handle passed as an argument.

(int) `GetTerminalType(`*terminal_handle*`)`
This will return a type code index for the terminal referenced by the handle passed as an argument.

This is not expected to be useful at present.

(int) `GetTerminalFlags(`*terminal_handle*`)`
This will return the flags for the terminal referenced by the handle passed as an argument.

This is not expected to be useful at present.

(int) `GetTerminalLocation(`*terminal_handle*`, `*array*`)`
This function returns the location for the terminal referenced by the handle passed as an argument. The second argument is an array of size two or larger which will receive the x-y coordinate, in microns. The function returns 1 on success, 0 otherwise.

(object_handle) `GetTerminalInstance(`*terminal_handle*`)`
This function returns a handle to the electrical subcircuit or device instance associated with the terminal referenced by the handle passed as an argument. This will not exist for formal terminals.

The returned object is an *electrical* object.

(int) `IsTerminalFormal(`*terminal_handle*`)`
This will return 1 if the terminal is "formal", i.e., it is one of the external connections to the current cell. Otherwise, 0 is returned if the terminal connects to internal objects only. The return value is -1 on error.

(object_handle) `GetTerminalObject(`*terminal_handle*`)`
This function returns a handle to a physical object that is associated with the terminal referenced by the handle passed as an argument. 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".

(int) `GetTerminalVgroup(`*terminal_handle*`)`
This will return the virtual group number for the terminal referenced by the handle passed as an argument. Terminals that are not associated with an object are "virtual" and represent pass-through between other cells or subcells. The returned value is -1 on error or if the terminal is not virtual.

(string) `GetTerminalLayer(`*terminal_handle*`)`
This returns a string containing the layer name for the terminal referenced by the handle passed as an argument. Non-virtual terminals are associated with a conducting layer.

D.5.3 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 the 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.

(object_handle) `ListGroupObjects(`*group*`)`
This function returns a handle to the list of objects in the current physical cell which constitute the group. The argument is the group index. The objects are copies, so can not be modified or selected. The objects returned have been processed by the "Conductor Exclude" directive, so may not precisely correspond to the "real" objects in the database. If an error occurs, 0 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.

(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.

(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 formal 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 formal terminals, the list will be empty.

D.5.4 Physical Devices

(device_handle) `ListPhysDevs(`*name*`, `*pref*`, `*index*`, `*l*`, `*b*`, `*r*`, `*t*`)`
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 extract. Either or both of these arguments can be null or empty, in which case no devices are excluded by the comparison. The third argument is the device index, which can be used to select a particular device. If this value is -1, any device index will be matched. The remaining four arguments set the corners, in microns, of the rectangular area in the cell to search. If all four values are zero, the entire cell is 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.

D.5.5 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 the 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. If an error occurs, -1 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.

(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.

D.5.6 Electrical Nodes

(int) `EnableNodeMap(`*on*`)`
This function sets whether or not node name mapping is used in netlist output from the electrical database. If the argument is nonzero, node mapping is set, and assigned node names will be used in output. Otherwise, the node names will be the internally assigned node numbers. This function returns 1 on success, 0 otherwise.

(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. When node mapping is enabled, netlist output will use the given string name rather than the node number. Since the node number is set internally, it can vary, whereas the given name is fixed. 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 the string will consist of the node number. This will return the mapped name whether or not node mapping is enabled. 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) `ListNodeTerminals(`*node*`)`
Return a handle to the list of terminals connected to the internal node number supplied as the argument.

(stringlist_handle) `ListNodeTerminalNames(`*node*`)`
This function returns a string list of the terminal names tied to the given node number. These come from the electrical database. If an error occurs, 0 is returned.

D.5.7 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.

This page intentionally left blank.

Index

! command, 303
!addcells command, 318
!area command, 335
!assemble command, 307
!bb command, 338
!bincnt command, 314
!bloat command, 333
!box2poly command, 332
!cc command, 347
!cd command, 310
!check command, 313
!checkgrid command, 312
!checkover command, 312
!clearall command, 329
!co command, 331
!compare command, 311
!cont command, 332
!desel command, 345
!display command, 323
!dr command, 339
!dumpfont command, 322
!dumplogofont command, 322
!dups command, 313
!empties command, 312
!errlayer command, 316
!errs command, 316
!exec command, 340
!exset command, 317
!fileinfo command, 315
!find command, 318
!fx command, 320
!fxcell command, 321
!gunzip command, 310
!gzip command, 310
!help command, 323
!helpfixed command, 324
!helpfont command, 324
!helpreset command, 324
!import command, 339
!join command, 335
!lal command, 328
!layer command, 325
!lisp command, 340
!listfuncs command, 343
!logfiles command, 311
!logofont command, 95, 322
!ltab command, 327
!ltsort command, 328
!manh command, 336
!mklib command, 329
!mmclear command, 330
!mmrecycle command, 329
!mmstats command, 330
!mo command, 330
!noacute command, 337
!origin command, 338
!path2poly command, 333
!poly2path command, 333
!polycheck command, 313
!polyfix command, 314
!polymanh command, 314
!polynum command, 314
!polyrev command, 337
!ptrms command, 319
!pwd command, 310
!rcq command, 332
!regen command, 313
!rehash command, 343
!rename command, 331
!rg command, 323
!rmfunc command, 343
!ro command, 331
!sa command, 331
!sced2xic command, 310
!script command, 340
!select command, 343
!set command, 346
!setcolor command, 322
!setflag command, 315
!sg command, 323
!showz command, 316
!source command, 316
!split command, 336
!sqdump command, 332

!summary command, 315
!svq command, 332
!sw command, 348
!symtab command, 346
!terms command, 319
!time command, 315
!tk command, 342
!togrid command, 337
!tospot command, 337
!unset command, 347
!ushow command, 319
!vmem command, 329
!xdepth command, 315
!zs command, 346
-B option, 6
-C option, 6
-C1 option, 6
-E option, 5
-G option, 5
-H option, 5
-K option, 5
-L option, 5
-S option, 6
-T option, 5, 349
.model lines, 414
.spinclude directive, 102
.splib directive, 102
.xicinit file, 14
.xicmacros file, 14, 157
.xicmacros file format, 158
.xicstart file, 14

example library file, 407
MultiNet keyword, 221

aborting commands, 23
About button, 71
abs function, 285
accelerator keys, 22
accelerators
    changing, 23
acos function, 285
acosh function, 285
AddCellProperty function, 565
AddError function, 501
AddLayer function, 537
AddLayerAlias function, 541
AddLayerGdsInMap function, 540
AddLayerGdsOutMap function, 540
AddLogMessage function, 502
AddMark function, 486
AddProperty function, 565
AddToBack variable, 134, 446
AdvanceZref function, 570
alias file, 169
Allocation button, 153
AltAxes keyword, 369
AltBackground keyword, 369
AltDriver keyword, 369
AltElecDriver keyword, 369
AltElecGridOnBottom keyword, 369
AltElecGridSpacing keyword, 369
AltElecGridStyle keyword, 369
AltElecResolution keyword, 369
AltElecShowGrid keyword, 369
AltFilled keyword, 356
AltGridOnBottom keyword, 369
AltGridSpacing keyword, 369
AltGridStyle keyword, 369
AltInvisible keyword, 359
AltPhysDriver keyword, 369
AltPhysGridOnBottom keyword, 369
AltPhysGridSpacing keyword, 369
AltPhysGridStyle keyword, 369
AltPhysResolution keyword, 369
AltPhysShowGrid keyword, 369
AltResolution keyword, 369
AltRGB keyword, 356
AltShowGrid keyword, 369
AndBits function, 500
AnyNoOverlap keyword, 196
AnyOverlap keyword, 194
arc button, 75
Arc function, 558
AreaHandle function, 543
ArrayDimension function, 499
ArrayDims function, 499
asin function, 285
asinh function, 285
AskConsoleReal function, 531
AskConsoleString function, 531
AskReal function, 531
AskString function, 531
assembly stream, 524
atan function, 285
atan2 function, 285
atanh function, 285
Attributes Menu, 155
AutoRename variable, 448
Axes keyword, 363

Background keyword, 367

Bandwidth button, 153
batch mode, 49
    start in, 6
batch mode commands, 50
Blink keyword, 359
blinking layers, 31
Bloat function, 557
BloatObjects function, 543
BloatZ function, 571
box, 38
    merge, 38, 178
box button, 38, 75
Box function, 557
BoxLineStyle keyword, 365
BoxZ function, 571
branch property, 426
break button, 76
BtnDown function, 529
BtnUp function, 529
button 1, 27
button 2, 30
button 3, 31
buttons, 27

Cadence compatibility, 59
Cadence technology file, 59
cap device, 80
Capacitance keyword, 224, 361
cccs device, 85
ccvs device, 85
ceil function, 285
cell arrays, 138
cell creation, 139
cell hierarchy, 39
cell info, 124
cell name alias file, 169
cell name mapping, 168, 179
Cell Names True Orient button, 161
Cell Properties button, 142
CELL PROPERTIES label, 395
cell rendering threshold, 161
cell scaling, 176
Cells List button, 122
cells panel, 122
    cell replacement, 123
    Clear button, 123
    Copy button, 123
    edit, 123
    info, 124
    listing, 122
    Master, 121, 123
    Rename, 124
    Search, 124
    Show, 124
    Tree, 123
CellsHandle function, 501
CellThreshold variable, 439
CGX format, 401
ChangeLayer function, 557
Check In Background button, 214
Check In Foreground button, 213
Check In Region button, 214
CheckForHoles function, 545
CheckObjectsConnected function, 545
CheckSolitary variable, 77, 445
Chg Layer button, 146
chlyr button, 43
CIF extensions, 398
    array, 399
    bound, 399
    cell name, 398
    labels, 398
    magnify, 399
    wires, 398
CIF format, 395
    box, 396
    comment, 396
    layer, 396
    polygon, 396
    symbol call, 397
    symbol definition, 396
    symbol termination, 396
    transformation, 397
    wire, 397
CIF: resolve layers control, 179
CifAddBBox variable, 453
CifLayerMode variable, 180, 451
CifOldWire variable, 453
CIFoutStyle variable, 177, 453
Clear Errors button, 212
Clear function, 483
ClearAll function, 484
ClearCell function, 482
ClearLayerAliases function, 541
ClearLayerGdsInMap function, 540
ClearSPtable function, 575
ClearTempLayer function, 573
ClipAround function, 554
ClipAroundAll function, 555
ClipAroundAllCopy function, 555
ClipAroundCopy function, 555
ClipIntersectCopy function, 557

ClipObjects function, 556
ClipTo function, 555
ClipToAll function, 556
ClipToAllCopy function, 556
ClipToCopy function, 556
ClipToGrid function, 536
Close function, 504
CloseArray function, 504
CloseLibrary function, 496
color panel, 163
colormap, 3
colormap options, 6
colors, 322
command line, 4
  options, 7
    -B, 6
    -C, 6
    -E, 5
    -G, 5
    -H, 5
    -K, 5
    -L, 5
    -S, 6
    -T, 5
    -d, 7
    -display, 7
    -name, 7
    -synchronous, 7
    --class, 7
    --display, 7
    --name, 7
    --no-xshm, 7
    --sync, 7
Comment keyword, 350
Commit function, 533
Compare Devices button, 248
compatibility
  sced, 310
  Strip For Export, 173
  terminal generation, 319
Compute Params button, 248
conductor groups, 239
Conductor keyword, 221, 360
configuration, 15
Connected keyword, 193
connecting devices, 43
connection points
  show, 86
connectivity test, 238, 239
consistency check, 313
  regeneration, 313
Constrain 45 button, 159
constrain 45s, 159
Constrain45 keyword, 364
Contact keyword, 223, 360
conversion, file format, 167
  alias file, 169
  cell names, 168
  layer suppression, 174
convert ascii to GDSII, 183
Convert button, 180
convert GDSII to text, 183
Convert Menu, 167
convert to CIF, 177
convert to GDSII, 177
convert to OASIS, 177
coordinate readout area, 22
Copy button, 144
copy cells, 123
Copy function, 561
copy objects, 27, 144
CopyToLayer function, 561
CoreSize function, 504
cos function, 285
cosh function, 285
crash, 16
Create Cell button, 139
create CGX hierarchy, 177
create Xic hierarchy, 177
CreateSymbol function, 562
Cross Section button, 151
CrsymOverwrite, 446
csw device, 85
CurCellBB function, 483
CurCellName function, 483
CurMode function, 497
current directory, 310
current layer, 20, 30
current transform, 73, 135
Current Transform button, 135
Cwd function, 526
CxCellBB function, 515
CxClearArea function, 515
CxClearSkipFlags function, 515
CxCompare function, 513
CxEdit function, 515
CxEstFlatMemoryUse function, 513
CxEstSkelSize function, 514
CxFileName function, 514
CxFileType function, 514
CxGetZlist function, 573
CxInfo function, 510

CxLayers function, 514
CxOpenFlat function, 517
CxOpenOdb function, 575
CxOpenZbdb function, 576
CxOpenZdb function, 576
CxSetArea function, 515
CxSetSkipEmpties function, 515
CxSetSkipFlag function, 515
CxSetupSkeleton function, 516
CxTopCells function, 514
CxWrite function, 516

DarkField keyword, 223, 361
DateString function, 526
decimal format layer names, 171
deck button, 47, 77
Decompose function, 557
default cell name conflict action, 134
Default when cells conflict button, 178
Define keyword, 351
Defined function, 504
Delete button, 144
delete cells, 123
Delete Empties function, 560
Delete function, 560
deleting objects, 38
Description keyword, 185
desel button, 77
Deselect function, 542
design rules, 193
    AnyNoOverlap, 196
    AnyOverlap, 194
    assignment, 210
    background check, 214
    browsing errors, 214
    check, 213
    check in region, 214
    clear errors, 212
    Connected, 193
    expressions, 190
    IfOverlap, 194
    interactive, 213
    level, 567
    limitations of tests, 210
    limits, 212
    MaxArea, 198
    MinArea, 198
    MinEdgeLength, 198
    MinNoOverlap, 203
    MinOverlap, 202
    MinSpace, 200
    MinSpaceFrom, 202
    MinSpaceTo, 200
    MinWidth, 199
    NoHoles, 193
    NoOverlap, 194
    Overlap, 193
    overlap definition, 211
    PartOverlap, 196
    print error, 215
    skip objects, 213
    state, 566
    vias, 211
DestroyDb function, 576
device block, 225
device library file, 407
    comments in, 408
    editing, 411
    example entry, 410
    syntax, 409
device menu, 40, 77, 443
device placement, 40
device properties, 44, 423
device.lib file, 407
devices, 41
devs button, 77
dio device, 82
directory change, 310
Display function, 485
DisplayAllText keyword, 364
Distance function, 568
DocsDir variable, 438
Don't convert invisible button, 174
Don't prompt for overwrite button, 178
donut button, 86
dots button, 44, 86
drag and drop, 19
drawing window, 19
Drc keyword, 362
DRCcheckArea function, 567
DRCcheckObjects function, 568
DRCgetLevel function, 567
DRCgetLimits function, 567
DrcInterMaxObjs keyword, 362
DrcInterMaxTime keyword, 362
DrcInterSkipInst keyword, 362
DrcMaxErrors keyword, 362
DrcNoPopup keyword, 362
DRCregisterExpr function, 568
DRCsetLevel function, 567
DRCsetLimits function, 567
DRCstate function, 566

DRCtestBox function, 568
DRCtestPoly function, 568
dual plane colorcells, 6
Dump Elec Netlist button, 245
Dump Error File button, 214
Dump LVS button, 246
Dump Phys Netlist button, 243
Dump to File command, 150
DumpElecNetlist function, 578
DumpLayerAliases function, 541
DumpPhysNetlist function, 577
DupArray function, 500

EdgeObjects function, 544
EdgesZ function, 571
Edit button, 131
Edit Extraction button, 261
Edit function, 481
Edit GDSII Keys button, 184
edit layer table, 165
Edit Layers button, 165
Edit Menu, 127
edit physical properties, 160
Edit Rules button, 215
edit stream i/o, 184
Edit Terminals button, 240
editing cells, 123, 131
editing context, 138
editing files, 121
editing properties, 140
editing subcells, 138
ElecAltBackground keyword, 369
ElecBackground keyword, 367
ElecDisplayAllText keyword, 364
ElecExpand keyword, 364
ElecGridOnBottom keyword, 363
ElecGridSpacing keyword, 363
ElecGridStyle keyword, 364
ElecLabelAllInstances keyword, 365
ElecLayer keyword, 354
ElecShowContext keyword, 365
ElecShowGrid keyword, 363
ElecShowTinyBB keyword, 365
ElecSnapping keyword, 363
Electrical button, 148
ELECTRICAL keyword, 400
electrical layers, 44
electrical mode, 37
    start in, 5
electrical mode editing, 39
electrical netlist, 245
Else keyword, 352
empty symbol, 179
EnableNodeMap function, 586
Endif keyword, 352
EnetBottomUp variable, 459
EnetSpice variable, 458
environment, 7
    DISPLAY, 7
    XEDITOR_FONT, 10
    XIC_DOCS_DIR, 10
    XIC_EXIT_CMD, 10
    XIC_GEOMETRY, 9
    XIC_HLP_PATH, 12
    XIC_LIB_PATH, 12
    XIC_LOGDIR, 10
    XIC_MENU_RIGHT, 10
    XIC_SCR_PATH, 12
    XIC_START_DIR, 10
    XIC_SYM_PATH, 12
    XIC_TMP_DIR, 9
    XT_AUTH_MODE, 8
    XT_LICENSE_PATH, 9
    XT_PREFIX, 9
    XT_SYSTEM_MALLOC, 11
    XT_USE_GTK_THEMES, 9
erase button, 39, 87
Erase function, 561
Erase Under button, 144
EraseBehindProps variable, 447
EraseBehindTerms variable, 457
EraseMark function, 487
EraseUnder function, 561
ErrorMsg function, 532
eval keyword, 351
EvalLayerExpr function, 572
Exec function, 497
exec keyword in scripts, 287
exit command, 10
Exit function, 486
exiting Xic, 125
exp function, 285
expand, 26
Expand button, 148
Expand function, 485
Expand keyword, 364
expanded view, 148
Export Control button, 173
exporting designs
    Strip For Export, 173
ExsetClear variable, 460
ExsetIncludeWireCap variable, 460

Extract RLC button, 249
ExtractAndSet function, 578
extraction, 217
    ground plane, 221
    methodology, 237
extraction, technology file entries, 221
extractNetResistance function, 580
ExtractOpaque variable, 457
extractRL function, 579

FastCap partition, 257
FastCap/FastHenry control function, 320
FastCap/FastHenry control panel, 258
FastCap/FastHenry interface, 249
FastCap/FastHenry interface setup, 250
FastHenry partition, 256
FastHenry terminals, 254
FcArgs variable, 261, 462
FcEdgeMax variable, 259, 462
FcPartMax variable, 259, 462
FcPath variable, 462
FcThinEdge variable, 259, 462
FhArgs variable, 261, 463
FhFreq variable, 261, 463
FhPath variable, 462
file compression, 310
file manager, 112
File menu, 111
file selection, 112
FileInfo function, 509
FileName function, 483
Files List button, 121
files panel, 121
    Contents, 121
    Edit, 121
fill panel, 164
fill patter editing, 164
Filled keyword, 356
FilterObjects function, 548
FilterObjectsA function, 548
find cells, 124
Find Terminals button, 241
FindPath function, 579
FixedFontName keyword, 366
FlatGenCount function, 554
FlatGenNext function, 554
FlatObjGen function, 553
FlatObjGenLayers function, 554
FlatObjList function, 553
FlatOverlapList function, 554
Flatten button, 139
Flatten function, 562
flatten hierarchy, 139, 562
FlattenPrefix variable, 219, 457
floor function, 285
font, 161
font file format, 93
Font1-6 keywords, 365
FontName keyword, 366
Format Conversion, 180
    clipping, 182
    empty cell removal, 182
    flattening, 182
    windowing, 182
format library, 235
FreeArray function, 504
Freeze Display button, 159
FreezeDisplay function, 486
FromArchive function, 518
FromCGX function, 518
FromCIF function, 518
FromGDS function, 518
FromOASIS function, 518
FromTxt function, 518
frozen window, 159
fullcursor script, 269
FullWinCursor variable, 439
function keys assignment, 361
function library, 264
functions, 340, 343
FxForeg variable, 260, 462
FxMinRectSize variable, 258, 461
FxPlaneBloat variable, 259, 461
FxUnits variable, 258, 462
F*N* keyword, 361

gauss function, 285
GDSII cell names, 177
GDSII extensions, 394
GDSII file format, 394
GDSII layer mapping, 172, 359, 454
GDSII to text conversion, 183
GDSII version number setting, 174
GDSII: map datatypes button, 179
GDSII: skip layers button, 174
GdsInToLower keyword, 362
GdsMunit variable, 454
GdsOutLevel variable, 454
GdsOutToUpper keyword, 363
gdstx button, 394
GenCells function, 501
GenLayers function, 538

GeomAnd function, 574
GeomAndNot function, 574
GeomCat function, 574
GeomNot function, 574
GeomOr function, 574
GeomXor, 574
Get function, 499
GetArchiveContext function, 509
GetButtonStatus function, 529
GetCellPrpHandle function, 564
GetCurAngle function, 533
GetCurLayer function, 537
GetCurLayerAlias function, 537
GetCurLayerDescr function, 537
GetCurMagn function, 534
GetCurMX function, 534
GetCurMY function, 534
GetDims function, 499
GetEdevObj function, 587
GetEdevProperty function, 587
GetError function, 501
GetGridInterval function, 535
GetGridOnTop function, 536
GetGridSnap function, 535
GetGridStyle function, 535
GetGroupBB function, 582
GetGroupCapacitance, 582
GetGroupName function, 582
GetGroupNetName function, 582
GetGroupNode function, 582
GetInstanceArray function, 552
GetInstanceName function, 553
GetInstanceXform function, 552
GetInstanceXformA function, 552
GetKey function, 531
GetLabelText function, 551
GetLabelXform function, 551
GetLastPrompt function, 497
GetLayerCap function, 539
GetLayerCapPerim function, 539
GetLayerEps function, 539
GetLayerLambda function, 539
GetLayerMinDimension function, 539
GetLayerResis function, 539
GetLayerRho function, 539
GetLayerThickness function, 539
GetLayerWireWidth function, 539
GetLogMessage function, 502
GetLogNumber function, 502
GetNodeGroup function, 586
GetNodeName function, 586
GetNodeNumber function, 586
GetNumberGroups function, 582
GetNumberNodes function, 586
GetObjectArea function, 548
GetObjectBB function, 548
GetObjectCoords function, 549
GetObjectFlags function, 549
GetObjectGroup function, 549
GetObjectID function, 548
GetObjectLayer function, 549
GetObjectMagn function, 550
GetObjectPerim function, 548
GetObjectsOdb function, 576
GetObjectState function, 549
GetObjectType function, 548
GetPdevBB function, 583
GetPdevContactBB function, 584
GetPdevContactDev function, 584
GetPdevContactDevIndex function, 584
GetPdevContactDevName function, 584
GetPdevContactGroup function, 584
GetPdevContactLayer function, 584
GetPdevContactName function, 584
GetPdevDual function, 583
GetPdevIndex function, 583
GetPdevMeasure function, 583
GetPdevName function, 583
getPID function, 530
GetProperty function, 565
GetPropertyString function, 565
GetPrpHandle function, 564
GetPscBB function, 585
GetPscContactGroup function, 585
GetPscContactName function, 585
GetPscContactSubc function, 585
GetPscContactSubcGroup function, 585
GetPscContactSubcIndex function, 586
GetPscContactSubcName function, 586
GetPscDual function, 585
GetPscIndex function, 585
GetPscName function, 585
GetSPdata function, 575
GetTechExt function, 498
GetTerminalFlags function, 581
GetTerminalInstance function, 581
GetTerminalLayer function, 582
GetTerminalLocation function, 581
GetTerminalName function, 581
GetTerminalObject function, 581
GetTerminalType function, 581
GetTerminalVgroup function, 581

GetWindow function, 484
GetWindowMode function, 485
GetWindowView function, 485
GetWirePoly function, 551
GetWireStyle function, 550
GetWireWidth function, 550
GetZlist function, 571
GetZlistDb function, 576
GetZref function, 570
Glob function, 527
global keyword, 289
global properties, 408
global variables, 289
gnd device, 43, 80
graphics support, 3
GRarc function, 490
GRbox function, 490
GRboxes function, 490
GRcheckError function, 489
GRclear function, 489
GRcopyDrawable function, 489
GRcreatePixmap function, 489
GRdefineColor function, 491
GRdefineFillpattern function, 492
GRdefineLinestyle function, 491
GRdestroyPixmap function, 489
GRdraw function, 489
GRgetDrawableSize function, 489
grid, 26, 323
grid registers, 323
grid style, 162
GridOnBottom keyword, 363
GridReg keyword, 364
GridSpacing keyword, 363
GridStyle keyword, 363
GRline function, 490
GRlines function, 490
GRopen function, 489
GroundPlane keyword, 221, 360
GroundPlaneClear keyword, 221, 360
GroundPlaneDark keyword, 221, 360
GroundPlaneGlobal variable, 457
GroundPlaneMethod variable, 222, 458
GroundPlaneMulti variable, 222, 458
Group function, 582
group number, 237
GroupObjects function, 545
GRpixel function, 490
GRpixels function, 490
GRpolygon function, 490
GRpolyLine function, 490
GRresetDrawable function, 489
GRsetBackground function, 491
GRsetColor function, 491
GRsetFillpattern function, 492
GRsetLinestyle function, 492
GRsetMode function, 492
GRsetWindowBackground function, 491
GRtext function, 491
GRtextExtent function, 491
GRupdate function, 492
GTK themes, 9

H function, 503
Halt function, 486
HandleArray function, 503
HandleCat function, 503
HandleContent function, 502
HandleDup function, 502
HandleDupNitems function, 503
HandleNext function, 502
HandlePurgeList function, 503
HandleReverse function, 503
HandleTruncate function, 502
hard copy driver names, 370
hard copy driver parameters, 370
hardcopy panel, 116
    Best Fit, 117
    format, 118
hardcopy plots, 116
HardCopyCommand keyword, 371
HardCopyDefHeight keyword, 372
HardCopyDefWidth keyword, 372
HardCopyDefXoff keyword, 372
HardCopyDefYoff keyword, 372
HardCopyDevice keyword, 370
HardCopyLegend keyword, 371
HardCopyMaxHeight keyword, 372
HardCopyMaxWidth keyword, 372
HardCopyMaxXoff keyword, 372
HardCopyMaxYoff keyword, 372
HardCopyMinHeight keyword, 372
HardCopyMinWidth keyword, 372
HardCopyMinXoff keyword, 372
HardCopyMinYoff keyword, 372
HardCopyOrient keyword, 371
HardCopyPixWidth keyword, 371
HardCopyResol keyword, 371
HCdump function, 494
HCerrorString function, 495
HCgetBestFit function, 493
HCgetDriver function, 493

HCgetLandscape function, 493
HCgetLegend function, 493
HCgetMetric function, 493
HCgetResol function, 493
HCgetResols function, 493
HCgetSize function, 494
HClistDrivers function, 492
HClistPrinters function, 495
HCmedia function, 495
hcopy button, 116
HCsetBestFit function, 493
HCsetDriver function, 492
HCsetGridInterval function, 494
HCsetGridOnTop function, 494
HCsetGridStyle function, 494
HCsetLandscape function, 493
HCsetLegend function, 493
HCsetMetric function, 493
HCsetResol function, 493
HCsetSize function, 494
HCshowAxes function, 494
HCshowGrid function, 494
help
  clear cache, 325
  fixed font, 324
  font, 324
Help button, 65
help database, 70
Help Menu, 65
help mode, 65
  path, 13
  file format, 415
  keyword input, 323
help path, 13
help viewer, 65
  Anchor Buttons, 69
  Anchor Highlight, 69
  Anchor Plain, 69
  anchor styles, 69
  Anchor Underline, 69
  back, 67
  Bad HTML Warnings, 69
  Clear Cache, 68
  cookies, 69
  Delayed Images, 69
  disk cache, 68
  Don't Cache, 68
  Find Text, 68
  forward, 67
  Freeze Animations, 69
  image formats, 69
  Log Transactions, 69
  No Cookies, 69
  No Images, 69
  Open, 67
  Open File, 67
  Print, 67
  Progressive Images, 69
  Quit, 67
  Reload, 67
  Reload Cache, 68
  Save, 67
  Search, 68
  Set Font, 68
  Show Cache, 69
  stop, 67
  Sync Images, 69
HelpMultiWin variable, 66, 463
HelpPath variable, 438
hex format layer names, 171
hierarchy of cells, 37, 562
HlpPath keyword, 354
HPGLfilled keyword, 356
HTML forms, 290
hypertext, 47, 89, 98
hypertext reference format, 431

iconic, 26
IfDef keyword, 352
IfnDef keyword, 352
IfOverlap keyword, 194
immediate execution, 287
Import Control button, 178
InCellNamePrefix variable, 450
InCellNameSuffix variable, 450
ind device, 81
INFINITY, 290
Info button, 152
InfoInternal variable, 124, 152, 447
initc property, 430
InitGen function, 500
initialization files, 14
InPath function, 505
int function, 285
interactive plotting, 87
interactive rule checking, 213
Intersect function, 569
InToLower keyword, 362
InToLower variable, 450
InToUpper variable, 450
InUseAlias variable, 450
inverting polarity, 108

Invisible keyword, 359
iplot button, 87
IsCellInMem function, 484
IsFileInMem function, 484
IsLayerConductor function, 538
IsLayerContact function, 538
IsLayerDarkField function, 538
IsLayerGround function, 538
IsLayerRouting function, 538
IsLayerVia function, 538
IsLayerVisible function, 538
IsPscContactIgnorable function, 585
isrc device, 81
IsTerminalFormal function, 581

jj device, 82
Join function, 557
JoinBreakClean variable, 441
JoinLimits function, 499
JoinMaxPolyGroup variable, 440
JoinMaxPolyQueue variable, 440
JoinMaxPolyVerts variable, 440
JoinObjects function, 545
Justify function, 559

KeepBadArchive variable, 452
Key Map button, 156
key mapping, 156
keyboard, 23
  !, 24
  arrow keys, 24, 88
  Bsp, 20, 23
  buffer, 23
  clear buffer, 26
  coordinate entry, 25
  Ctrl, 26
  Ctrl-A, 20, 25
  Ctrl-B, 214
  Ctrl-C, 25, 213
  Ctrl-E, 20, 25
  Ctrl-F, 214
  Ctrl-G, 26, 162
  Ctrl-N, 26
  Ctrl-P, 26, 214
  Ctrl-R, 26
  Ctrl-U, 20, 23, 26
  Ctrl-V, 26
  Ctrl-X, 26
  Ctrl-Z, 26
  Delete, 24
  Esc, 20, 23
  function keys, 25
  Home, 24
  interrupt, 25
  numeric +, 24
  numeric -, 24
  Page Down, 24, 214
  Page Up, 24, 214
  Shift, 26
  Shift-Tab, 24
  Tab, 24
KeyDown function, 529
keypress buffer, 22
KeyUp function, 530

label button, 47, 88
label editing, 88
label font, 93
Label function, 558
Label True Orient button, 160
LabelAllInstances keyword, 364
labloc property, 425
labrf property, 428
lambda, 351
layer aliasing, 171
layer button, 93
layer change of object, 146
layer expressions, 190
Layer expressions, functions, 191
layer filtering, 171
Layer function, 562
layer names, 170
layer table, 20
layer visibility, 30
layer-specific, 27, 93
LayerAlias function, 541
LayerHandle function, 537
LayerList variable, 449
LayerName keyword, 354
LayerPartSize variable, 441
LayerSearchUp variable, 238, 239, 457
LayerTable variable, 439
layout vs. schematic, 246
LibPath keyword, 353
LibPath variable, 438
Libraries List button, 121
library file, 264
library files, 406
library path, 13, 349
LibSave property, 408
Lisp language, 341
Lisp parser, 341

ListAddBack function, 506
ListAddFront function, 506
ListAlphaSort function, 506
ListCellsInMem function, 484
ListConcat function, 506
ListContent function, 506
ListDirectory function, 528
ListElecDevs function, 587
ListFormatCols function, 506
ListFunctions function, 497
ListGroupDevContacts function, 582
ListGroupObjects function, 582
ListGroupSubcContacts function, 582
ListGroupTerminalNames function, 583
ListGroupTerminals function, 583
ListHandle function, 505
ListIncluded function, 506
listing cells, 122
listing files, 121
listing libraries, 121
ListModCellsInMem function, 484
ListNext function, 506
ListNodeTerminalNames function, 586
ListNodeTerminals function, 586
ListPdevContacts function, 584
ListPdevMeasures function, 584
ListPhysDevs function, 583
ListPhysSubckts function, 584
ListPscContacts function, 585
ListReverse function, 506
ListTopCellsInMem function, 484
ListTopFilesInMem function, 484
ListUnique function, 506
ln function, 285
loading rawfiles, 101
LockMode variable, 447
log files, 10
log function, 285
logfiles, 15
logo button, 95
    use file, 443
logo font, 93
Logo function, 559
LogoEndStyle variable, 96, 442
LogoManhattan variable, 95, 322, 442
LogoPathWidth variable, 96, 442
LogoPretty variable, 95, 322, 442
LogoPrettyFont variable, 95, 322, 443
LogoToFile variable, 96, 443
long text labels, 90
LongName keyword, 184
ltra device, 84
LVS, 237, 246

macros, 156
mail client, 35
MakeObjectCopy function, 543
Manhattanize function, 557
ManhattanizeObjects function, 544
ManhattanizeZ function, 572
Master button, 41
master cells, 123
master menu length, 445
MasterMenuLength variable, 137, 445
MaxArea keyword, 198
MaxDistObjToObj function, 569
MaxDistPointToObj function, 569
MaxPrpLabelLen variable, 92, 439
MaxRoundSides variable, 443
memory, 153
memory management, 11
menu
    right side placement, 10
Merge boxes button, 178
Merge Control pop-up, 134
MergeInput variable, 178, 448
MergeOnRead keyword, 362
Message function, 532
MilliSec function, 526
MinArea keyword, 198
MinDistObjToObj function, 569
MinDistPointToObj function, 568
MinDistPointToSeg function, 568
MinDistSegToObj function, 568
MinEdgeLength keyword, 198
MinNoOverlap keyword, 203
MinOverlap keyword, 202
MinSpace keyword, 200
MinSpaceFrom keyword, 202
MinSpaceTo keyword, 200
MinWidth and logo text, 95
MinWidth keyword, 199
Mode function, 496
mode switch, 148
model library file, 42, 414
model property, 128, 429
model.lib file, 414
models subdirectories, 414
ModifyTerminal function, 580
mos devices, 41
mos substrate bias, 42
Move button, 144

Move function, 561
move objects, 27, 144
MoveToLayer function, 561
mut device, 81
mutlrf property, 428
mutual inductors, 81
mutual property, 426

name property, 46, 128, 425
native file format, 400
    CIF extensions, 398
    Property extension, 399
NDRC layer, 213
netlist, 245
    electrical, 245
    physical, 243
netlist extraction, 217
New button, 111
NewCellName function, 482
NewCurLayer function, 537
njf device, 82
nmes device, 84
nmos device, 83
nmos1 device, 83
No Pop Up errors button, 213
NoAlt keyword, 371
NoAltSelection variable, 28, 442
NoAskOverwrite variable, 134, 178, 448
NoAsyncSpice variable, 101, 444
NoCheckEmpties variable, 132, 179, 312, 449
NoCoincCheck variable, 179, 449
NoConstrainRound variable, 443
node mapping, 97
node mapping editor, 46
node naming, 46
node property, 424
nodemap property, 428
NoDisplayCache variable, 439
nodmp button, 46, 97
NoDRC flag, 96
NoDrcDataType keyword, 185
NoDrcDatatype keyword, 359
NoDriverLabels variable, 445
NoEnet variable, 458
NoExsetAllDevs variable, 460
NoExsetCreate variable, 460
NoGdsMapOk variable, 177, 454
NoHelpDefault variable, 65, 463
NoHoles keyword, 193
NoMapDatatypes variable, 179, 451
NoMeasure variable, 458
NoMergeBoxes variable, 178, 443
NoMergeParallel variable, 227, 458
NoMergeSeries variable, 227, 458
NoOverlap keyword, 194
NoOverwriteElec variable, 134, 448
NoOverwritePhys variable, 134, 448
NoPermute variable, 460
nophys property, 430
NoPixmapStore variable, 439
NoPnet variable, 459
NoPnetDevs variable, 459
NoPnetSpice variable, 459
NoPolyCheck variable, 179, 449
NoPopUpLog variable, 448
NoReadExclusive variable, 134, 445
NoReadLabels variable, 449, 515
NoSnapToEdges variable, 441
NoSpiceTools variable, 445
NotBits function, 500
NoWireWidthMag, 446
npn device, 82
NULL, 290
NumCellsInMem function, 484
NumHandles function, 502
NumToTxtCells variable, 451

OASIS text: no line wrap button, 181
OASIS text: print offsets button, 181
OASIS: find repetitions button, 175, 181
OASIS: string indirection button, 175, 181
OASIS: use compression button, 174, 181
OASIS: write crc checksum button, 175, 181
OasPrintNoWrap variable, 451
OasPrintOffset variable, 451
OasReadNoChecksum variable, 451
OasWriteChecksum variable, 456
OasWriteCompressed variable, 454
OasWriteNameTab variable, 455
OasWriteNoGCDcheck variable, 457
OasWriteNoTrapezoids variable, 456
OasWriteRep variable, 455
OasWriteWireToBox variable, 456
object breaking, 76
object copy, 561
object copying, 144
object creation
    arcs, 75
    boxes, 75, 557
    disks and ellipses, 100
    donut, 86
    labels, 88, 558

polygons, 99, 558
shapes templates, 103
wires, 107, 558
object deletion, 24, 38, 144, 560
object erasing, 87, 561
object info, 152
object invert, 561
object move, 561
object movement, 144
object polarity inversion, 108
object rotation, 104, 562
object stretching, 145
ObjectCopy function, 547
ObjectCopyFromString function, 547
ObjectCopyToLayer function, 547
ObjectDelete function, 546
ObjectDeselect function, 547
ObjectHandleDup function, 546
ObjectHandlePurge function, 546
ObjectMove function, 547
ObjectMoveToLayer function, 547
ObjectNext function, 546
ObjectSelect function, 546
ObjectString function, 547
ObjectZ function, 571
old mutual property, 426
opamp device, 85
Open button, 111
open file dialog, 114
Open function, 527
OpenCell function, 482
OpenLibrary function, 496
OrBits function, 500
other property, 128, 430
OutCellNamePrefix variable, 453
OutCellNameSuffix variable, 453
OutToLower variable, 452
OutToUpper keyword, 363
OutToUpper variable, 452
OutUseAlias variable, 452
Overlap keyword, 193

panic, 16
panning, 24, 30
param property, 128, 430
parameter setting, 437
parameter value comparison, 248
parameter value extraction, 248
parameterized cells, 55
ParseLayerExpr function, 572
PartOverlap keyword, 196
Path keyword, 353
Path variable, 438
paths, 11
help, 13
library, 13
script, 13
symbol, 12
updating, 12
PathToEnd function, 504
PathToFront function, 505
PCL, 118
Peek button, 151
peek mode, 149
PeekSleepMsec variable, 151, 447
PhysAltBackground keyword, 369
PhysBackground keyword, 367
PhysDisplayAllText keyword, 364
PhysExpand keyword, 364
PhysGridOnBottom keyword, 363
PhysGridOrigin variable, 440
PhysGridSpacing keyword, 363
PhysGridStyle keyword, 364
Physical button, 148
PHYSICAL keyword, 400
physical mode, 37
switch to, 148
physical netlist, 243
physical text, 95
PhysLabelAllInstances keyword, 365
PhysPropTextSize variable, 447
PhysShowContext keyword, 365
PhysShowGrid keyword, 363
PhysShowTinyBB keyword, 365
PhysSnapping keyword, 363
PictorialDevs variable, 77, 443
pjf device, 82
Place button, 136
place cells, 136
Place function, 560
place panel, 136
cell reference point, 137
Replace, 138
Use Array, 138
PlaceParams function, 560
PlaceTemplateArgs function, 560
PLEX record, 395
plot button, 47, 98
plot to file, 117
pmes device, 84
pmos device, 83
pmos1 device, 83

PnetBottomUp variable, 459
PnetIncludeWireCap variable, 459
PnetListAll variable, 219, 459
PnetShowGeometry variable, 459
pnp device, 82
Point function, 530
point operation, 27
polyg button, 38, 99
Polygon function, 558
polygon vertex editor, 99
polygons, 38
pop button, 138
Pop function, 482
Popen function, 527
PopGhost function, 488
Postscript, 118
preferences script, 269
PressButton function, 529
Print function, 532
print help text, 67
PrintLog function, 532
PrintString function, 532
PrintStringEsc function, 532
prompt line, 19
properties, 127, 139
  adding, 44
  addition of, 141
  branch, 426
  changing, 47
  gdswire, 422
  global, 421
  grid, 421
  iplot, 421
  labloc, 425
  labrf, 428
  model, 429
  mutlrf, 428
  mutual, 426
  name, 425
  node, 424
  nodemap, 428
  nophys, 430
  old mutual, 426
  other, 430
  param, 430
  pathtype, 422
  plot, 421
  run, 421
  symbolic, 428
  text, 422
  use in SPICE output, 128
  value, 429
Properties button, 139
properties panel
  Delete, 142
  Edit, 140
property deletion, 142
Property Editor, 44
Property Editor window, 139
Property Info window, 140
prpty button, 44
PrptyAdd function, 563
PrptyHandle function, 563
PrptyNext function, 563
PrptyNumber function, 563
PrptyRemove function, 564
PrptyString function, 563
pseudo-flat representation, 192
pseudo-properties, 129
PSlineWidth variable, 445
Push button, 138
Push function, 482
PushGhost function, 487
PushGhostBox function, 487
put button, 100
Put function, 561
Pwd function, 526

QpathGroundPlane variable, 238, 458
Query Errors button, 215
Quick Paths button, 238
Quit button, 125

random function, 285
Read Into Current button, 184
ReadArchiveContext function, 510
ReadChar function, 527
ReadLayerAliases function, 541
ReadLine function, 527
ReadSPtable function, 574
RecallArchiveContext function, 510
RecallTransform function, 533
redo, 24, 39, 533
Redo button, 143
redo button, 39
Redo function, 533
Redraw function, 486
redraw screen, 26
RegCompare function, 505
RegCompile function, 505
RegError function, 505
Release Notes button, 71

RemoveCellProperty function, 566
RemoveCurLayerExKeyword function, 537
RemoveLayer function, 537
RemoveLayerAlias function, 541
RemoveLayerGdsOutMap function, 540
RemovePath function, 505
RemoveProperty function, 566
rename cells, 124
RenameCell function, 483
RenameLayer function, 537
renaming cells, 331
RepartitionZ function, 572
replace cells, 138
Replace function, 560
report execution time, 153
res device, 80
Resistance keyword, 224, 361
RESOLUTION keyword, 400
resources, 322
RGB keyword, 355
Rho keyword, 224, 361
rint function, 285
RLC extraction control panel, 258
RLSolverDelta variable, 234, 461
RLSolverGridPoints variable, 234, 461
RLSolverMaxPoints variable, 234, 461
RLSolverTryTile variable, 234, 461
Rotate function, 562
RotateToLayer function, 562
round button, 100
round figure sides, 104
RoundFlashSides keyword, 365
Routing keyword, 360
Rulers button, 152
RulerSnapToGrid variable, 447
run button, 47, 101
running SPICE, 101
    output to file, 101

Save As button, 115
Save button, 114
save file dialog, 114
Save function, 483
save help text, 67
Save Tech button, 15, 155, 349
SaveArchiveContext function, 510
SaveCellAsNative function, 519
saving cells, 114, 115
SCED layer, 43, 354
screen layout, 17
script labels, 91
script path, 13
ScriptPath keyword, 264, 354
ScriptPath variable, 438
scripts, 264
    #define, 285
    #else, 285
    #endif, 285
    #ifdef, 285
    #ifndef, 285
    #macro, 285
    arrays, 275
    break, 283
    char constants, 273
    continue, 284
    data types, 272
    debugging, 266
        breakpoints, 267
        Execute menu, 267
        execution, 267
        File menu, 267
        monitor, 268
        open, 267
        print, 267
        reset, 267
        single-stepping, 267
    dowhile, 283
    editing, 267
    elif, 282
    error reporting, 272
    forms interface, 295
    goto label, 284
    hex constants, 273
    if elif else end, 282
    math functions, 285
    new script, 267
    operators, 279
    predefined constants, 290
    rehash, 268
    repeat, 283
    scalars, 273
    string subscripts, 274
    strings, 273
    variable types, 272
    while, 283
scripts, from prompt line, 303
search help database, 68
search paths, 11
seed function, 285
Select function, 542
Select Groups button, 238
SelectHandle function, 543

SelectHandleTypes function, 543
selections, 27, 38
    !select command, 343
    associated labels, 25, 26
    deselection, 77
    hierarchy, 28
SelectLast function, 542
SelectTime variable, 442
SepString function, 504
server mode, 50
    ascii protocol, 53
    binary protocol, 54
    start in, 6
Set Color button, 163
Set Fill button, 164
Set Flags button, 213
Set Font button, 161
Set function, 498
Set Grid button, 162
Set Interactive button, 213
Set keyword, 351
set layer colors, 163
Set Limits button, 212
SetButtonStatus function, 528
SetConvertArea function, 521
SetConvertFlags function, 521
SetConvertScale function, 517
SetCurLayer function, 536
SetCurLayerAlias function, 536
SetCurLayerDescr function, 536
SetCurLayerExKeyword function, 536
SetEdevProperty function, 587
SetExpand function, 498
SetGrid function, 535
SetGridOnTop function, 536
SetGridStyle function, 535
SetIndent function, 531
SetInstanceArray function, 552
SetInstanceName function, 553
SetInstanceXform function, 552
SetInstanceXformA function, 552
SetKey function, 498
SetLabelText function, 551
SetLabelXform function, 551
SetLayerNoDRCdatatype function, 540
SetLayerSpecific function, 541
SetLayerVisible function, 538
SetMapToLower function, 518
SetMapToUpper function, 518
SetMergeInRead function, 521
SetNodeName function, 586
SetObjectBB function, 549
SetObjectCoords function, 550
SetObjectLayer function, 549
SetObjectMagn function, 550
SetPrintLimits function, 532
SetSelectMode function, 542
SetSkipInvisLayers function, 519
SetSPdata function, 575
SetStripForExport function, 519
SetTechExt function, 498
SetTransform function, 533
SetWireStyle function, 550
SetWireToPoly function, 551
SetWireWidth function, 550
SetZref function, 570
sgn function, 285
shape templates
    and, 103
    arc, 103
    box, 103
    dot, 103
    or, 103
    poly, 103
    tri, 103
    ttri, 103
shapes button, 103
shapes templates, 103
shell escape, 303
Shell function, 530
Shell variable, 441
ShiftBits function, 500
show cell names, 160
Show Cell Names button, 160
Show Context button, 160
Show Errors button, 214
Show Groups button, 239
show instance origin, 161
show labels, 160
Show Labels button, 160
Show Location command, 150
Show Nodes button, 239
Show Origin Mark button, 161
Show Paths button, 239
show physical properties, 160
Show Properties button, 160
show rulers, 152
show subcell context, 160
show terminals, 107
Show Terminals button, 240
Show Tree button, 125
ShowAxes function, 535

ShowContext keyword, 365
ShowDb function, 576
ShowDots variable, 443
ShowGhost function, 488
ShowGrid function, 535
ShowGrid keyword, 363
ShowInstanceMark keyword, 365
ShowPhysProps keyword, 364
ShowPrompt function, 531
ShowSymbolic function, 486
ShowTinyBB keyword, 365
side menu, 73
sides button, 104
Sides function, 558
sin function, 285
sinh function, 285
Sizeof function, 509
SkelClear function, 522
SkelClearGeom function, 522
Skeleton Mode, 185
SkeletonMode variable, 185, 451
SkelReadDigest function, 523
SkelReadGeom function, 522
SkelSaveNum variable, 451
SkelSetDataWin function, 521
SkelWriteDigest function, 523
SkelWriteGeom function, 522
SkelWriteGeomSplit function, 523
Skip testing coincident button, 179
Skip testing empty button, 179
Skip testing polygons button, 179
SkipInvisible variable, 452
snap grid, 162
SNAPNODE record, 395
Snapping keyword, 363
Sopen function, 527
SortArray function, 500
Source Physical button, 242
Source SPICE button, 241
SourceAllDevs variable, 459
SourceClear variable, 460
SourceCreate variable, 460
SourceSpice function, 578
spice analysis, 47
SPICE deck creation, 77
spice device line, 429
spice key mapping, 444
SPICE output, 47
SPICE plots, 98
SpiceAlias variable, 444
SpiceExecDir variable, 444
SpiceExecName variable, 444
SpiceHost variable, 444
SpiceListAll variable, 444
SpiceProg variable, 444
spicetext label, 47, 89
spin button, 104
spiral script, 269
spiralform script, 269
Split function, 562
SplitObjects function, 545
spot size, 337, 441
SpotSize variable, 86, 101, 441
sqrt function, 285
starting *Xic*, 3
    no technology file, 349
StartTiming function, 526
static keyword, 288
static variables, 288
status area, 21
StopTiming function, 526
StoreTransform function, 533
Strcasecmp function, 507
Strcat function, 507
strch button, 39
Strchr function, 508
Strcmp function, 507
Strdup function, 507
StreamData keyword, 359
StreamIn keyword, 185, 359
StreamInstance function, 525
StreamOpen function, 524
StreamOut keyword, 185, 359
StreamRun function, 526
StreamSource function, 524
StreamTopCell function, 524
Stretch button, 145
StringHandle function, 505
strip button, 393
Strip For Export button, 394
StripForExport button, 173
StripForExport variable, 452
Strlen function, 509
Strncasecmp function, 507
Strncmp function, 507
Strpath function, 508
Strrchr function, 508
Strstr function, 508
Strtok function, 507
StuffText function, 497
style button, 104
subcircuit creation, 45

subcircuit placement, 41
subcircuit terminals, 45, 105
subct button, 105
SubstrateEps keyword, 362
SubstrateEps parameter, 262
subwindows, 150
sw device, 85
symbl button, 45, 106
symbol names, 394
symbol path, 12
    updating, 12
symbolic cells
    expanded view, 150
Symbolic keyword, 223, 361
symbolic mode, 106
symbolic property, 428
symbolic representation, 45
System function, 530

tan function, 285
tanh function, 285
Tcl/Tk, 342
Techfile function, 498
technology file, 15, 349
    !set, 353
    backslash continuation, 349
    eval, 351
    example, 372
    extension, 5, 349
    macros, 351
    scripts in, 354
TeePrompt variable, 439
temp directory, 9
TempFile function, 528
template cells, 55
TermDefault keyword, 221, 360
TermMarkSize variable, 457
terms button, 107
TermTextSize variable, 457
TestCoverage function, 572
text editor, 32
Text Editor button, 184
text to GDSII conversion, 183
TextCmd function, 497
TextWindow function, 532
themes, 9
Thickness keyword, 224, 361
Tiny Boxes button, 161
To CGX button, 177
To CIF button, 177
To GDSII button, 177
To OASIS button, 177
To Xic button, 177
ToCGX conversion, 177
ToCGX function, 519
ToChar function, 508
ToCIF conversion, 177
ToCIF function, 519
ToFormat function, 508
ToGDS function, 519
ToGDSII conversion, 177
ToGdsLibrary function, 520
ToOASIS conversion, 177
ToOASIS function, 520
toolbar menus, 22
TopCellName function, 483
ToReal function, 508
ToSpice function, 520
ToString function, 508
ToTxt function, 520
ToXIC conversion, 177
ToXIC function, 519
tra device, 84
Tranline keyword, 224, 361
TransformZ function, 571
tree diagram, 123, 125
txgds button, 394
TypeOf function, 504

undo, 24, 39, 143, 533
    list length, 143
Undo button, 143
undo button, 39
Undo function, 533
UndoListLength variable, 446
unerase, 100
Unset function, 498
update technology file, 15
UpdateNative function, 483
updating technology file, 155
urc device, 84
UseLayerAlias variable, 450
UseLayerList variable, 449
user-definable commands, 264
UseTransform function, 534

value property, 128, 429
variables, 346
    '!' commands, 440
    Attributes Menu commands, 447
    Edit Menu commands, 445
    extraction menu commands, 457

fastcap, 461
fasthenry, 461
general visual, 439
help, 463
paths, 438
printing, 445
selections, 442
setting, 346
side menu commands, 442
special constructs, 437
SPICE interface, 444
unsetting, 347
View Menu commands, 447
vcc device, 43, 80
vccs device, 84
vcvs device, 85
Versatec, 118
version, 26
VersionString function, 499
vertex editor, 44
via expression, 222
Via keyword, 222, 360
view, 24, 147
save, 26
View button, 147
View Menu, 147
viewing huge cells, 159
Viewport button, 150
viewports, 150
virtual terminals, 106
vsrc device, 82

window cancellation, 31
Window function, 484
wire button, 38, 107
wire connections, 43
wire end style setting, 104
Wire function, 558
wire vertex editor, 108
wire width setting, 104
wires, 38
convert to polygons, 100
WireWidth keyword, 359
WordPerfect, 118
WpFilled keyword, 357
WR button, 35
wr_lp script, 120
WriteArchiveContext function, 510
WriteChar function, 528
WriteGeomSplit function, 520
WriteLine function, 527

X display
specification, 7
XfigFilled keyword, 357
xic_error.log file, 15
xic_font file, 14
XIC_HLP_PATH, 415
xic_keymap file, 156
XIC_LIB_PATH, 354
xic_logofont file, 14
xic_mem_errors.log file, 16
xic_run.log file, 15
XIC_SCR_PATH, 264
xic_stipples file, 15
xic_tech file, 5, 349
XIC_TMP_DIR environment variable, 116
xor button, 108
Xor function, 561
XorBits function, 500
XSectThickness variable, 447

Yank function, 561
yank script, 269
yank/put, 100

Zarea function, 570
ZfromFile function, 573
Zhead function, 570
Zlength function, 570
ZoidZ function, 571
Zoom button, 150
zooming, 24, 31, 150
ZtoFile function, 573
ZtoObjects function, 572
ZtoTempLayer function, 573
Zvalues function, 570

This page intentionally left blank.