Graph
Pak

GraphPak for OSF Motif

Programmer’s Reference
Version 2.1

ICS

Integrated Computer
Solutions, Incorporated

Integrated Computer Solutions, Inc.
54 B Middlesex Turnpike, Bedford, MA 01730
Tel: 617.621.0060
Fax: 617.621.9555
E-mail: info@ics.com

Trademarks

Second Printing
January 2009

Table of Contents

Page

Preface xvii

1. Introduction: XiPlotter Architecture 3

1.1 XiPlotter Architecture 4
1.2 XiPlotter Data Types 7
1.3 Plot Data Representation 8

2. Hello, World 11

2.1 The Program 12
2.2 Compiling and Linking the Program 16

3. Plotter Data Types and Resources 19

3.1 Double Resources 20
3.2 String And XmString Resources 21
3.3 Fonts and Displayed Text 22
3.3.1 Font List Inheritance 22
3.3.2 Automatically Created XmStrings 23
3.3.3 PostScript Font Lists 26
3.3.4 Internationalized Text Display 28
3.4 Markers 29
3.5 Fill Patterns 31
3.6 Line Patterns 33
3.7 Line Widths 35
3.8 Colors 36
3.9 Cursors 37
3.10 Inherited Resource Values 38

4. Plot Data Representation 39

4.1 The XiPlotData Abstraction 40
4.2 Data in Arrays of Values 44
4.3 Data in Arrays of Structures 45
4.4 Data of Mixed Types 46
4.5 Linear Data 47
4.6 Linear Transformations of Data 48
4.7 String Data 49

5. The Plotter Widget 51

5.1 Creating an XiPlotter Widget 52
5.2 Plotter Resources 53
    5.2.1 Default Colors and Background Resources 56
    5.2.2 Fonts 57
    5.2.3 The Axes 59
    5.2.4 The Title 60
    5.2.5 The Legend 61
    5.2.6 Plotter Layout 64
    5.2.7 Double Buffering and Backing Store 66
    5.2.8 Interaction Resources and Callbacks 67
    5.2.9 Cursor Resources 67
    5.2.10 Printing Resources 69
    5.2.11 Constraint Resources 69
5.3 Plotter Redisplay 71
5.4 Printing the Plotter Widget 72
    5.4.1 Printout Size and Orientation 72
    5.4.2 Printout Scale 74
    5.4.3 Spooling a PostScript File to a Printer 75

6. The Axis Widget 77

6.1 Querying the Axes of a Plotter 79
6.2 Axis Resources 82
    6.2.1 The Title 82
    6.2.2 The Labels 84
    6.2.3 The Axis and Tick Marks 85

- 6.2.4 The Frame and the Origin 87
- 6.2.5 The Grid 88
- 6.2.6 Axis Layout 89
- 6.3 Controlling Axis Numbering 90
- 6.3.1 Computing Axis Bounds Automatically 91
- 6.3.2 Setting Axis Bounds Manually 92
- 6.3.3 Setting Secondary Y Axis Bounds 93
- 6.3.4 Specifying Logarithmic Axes 96
- 6.3.5 Computing Axis Marks Automatically 97
- 6.3.6 Specifying Axis Marks Explicitly 98
- 6.3.7 Computing Axis Labels Automatically 101
- 6.3.8 Specifying Axis Labels Explicitly 102
- 6.4 World <--> Pixel Coordinate Conversion 105

7. The LinePlot Widget 109

- 7.1 Creating a LinePlot 110
- 7.2 LinePlot Resources 110
- 7.2.1 Lines 111
- 7.2.2 Markers 112
- 7.2.3 Fill Patterns 114
- 7.2.4 Impulses 115
- 7.2.5 Shadows 115
- 7.2.6 Data Resources 116
- 7.3 Specifying LinePlot Data 117
- 7.3.1 Arrays of Doubles, Floats, or Ints 117
- 7.3.2 Specifying Data in Structures 117
- 7.3.3 Changing Data in Place 120
- 7.3.4 Adding Data Points Dynamically 120

8. The ErrorPlot Widget 125

- 8.1 Creating an ErrorPlot 126
- 8.2 ErrorPlot Resources 126
- 8.2.1 Error Bar Resources 127
- 8.2.2 Fixed Error Resources 128
- 8.2.3 Variable Error Resources 129
- 8.3 Specifying ErrorPlot Data 129

8.3.1 Arrays of Doubles, Floats, or Ints 130
8.3.2 Specifying Error Values from Structures 130
8.3.3 Changing Data in Place 132

9. The BarPlot Widget 135

9.1 Creating a BarPlot 136
9.2 BarPlot Resources 136
9.2.1 Bar Color and Pattern 137
9.2.2 Bar Outline 138
9.2.3 Bar Shadows 139
9.2.4 Bar Positioning, Grouping and Stacking 140
9.2.5 Data Resources 142
9.3 Specifying BarPlot Data 143
9.3.1 Arrays of Doubles, Floats, and Ints 143
9.3.2 Specifying Data in Structures 145
9.3.3 Changing Data in Place 145

10. The HistoPlot Widget 147

10.1 Creating a HistoPlot 148
10.2 HistoPlot Resources 148
10.2.1 Boundary and Bin Resources 149
10.2.2 Data Resources 150
10.3 Specifying HistoPlot Data 151
10.3.1 Arrays of Doubles, Floats, and Ints 151
10.3.2 Specifying Data from Structures 151
10.3.3 Changing Data in Place 152

11. The PiePlot Widget 155

11.1 Creating PiePlot Widgets 156
11.2 PiePlot Resources 159
11.2.1 PiePlot Fill Pattern 159
11.2.2 PiePlot Outline 160
11.2.3 PiePlot Shadows 160
11.2.4 PiePlot Labels 161

11.2.5 PiePlot Data 163
11.2.6 PiePlot Positioning 164

12. The HighLowPlot Widget 167

12.1 Creating a HighLowPlot 168
12.2 HighLowPlot Resources 170
12.2.1 Visual Resources 170
12.2.2 Data Resources 171
12.3 Specifying XiHighLowPlot Data 172
12.3.1 Arrays of Doubles, Floats, or Ints 172
12.3.2 Using Arrays of Structures 174
12.3.3 Notifying the Widget of Changed Data 176

13. Annotating Plots 179

13.1 Creating Annotations 180
13.2 TextPlot and ImagePlot Resources 182
13.2.1 Annotation Position 183
13.2.2 Annotation Offsets 185
13.2.3 Annotation Size and Margins 187
13.2.4 Annotation Colors 187
13.2.5 Arrows 188
13.2.6 Shadows and Trim 189
13.2.7 TextPlot Resources 190
13.2.8 ImagePlot Resources 192
13.3 Positioning Annotations 193
13.3.1 Querying LinePlot and HighLowPlot values 194
13.3.2 Querying the Bounds of a BarPlot Bar 195
13.3.3 Querying the Bounds of a PiePlot Wedge 196
13.3.4 Example: Annotating a PiePlot 198
13.4 Updating Image Annotations 199

14. The FadePlot Widget 201

14.1 FadePlot Resources 202
14.2 Using the FadePlot 204

14.2.1 The FadePlot as Backdrop and Background 206
14.2.2 Non-Linear Color Interpolation 207
14.2.3 Note on Color Usage 208

15. Drawing Your Own: the CallbackPlot 211

15.1 CallbackPlot Resources 212
15.1.1 Untyped Data Resources 213
15.1.2 Callback Resources 214
15.1.3 Control Resources 220
15.2 CallbackPlot Example 221

16. User Interaction with the Plotter 227

16.1 Selecting Items in the Legend 228
16.1.1 Example: Highlighting a Selected Plot 229
16.2 Selecting Points 230
16.2.1 Example: Hit Detection 232
16.3 Crosshairs and Mouse Motion 233
16.3.1 Example: Displaying Mouse Coordinates 235
16.4 Selecting Regions 236
16.4.1 Example: Zooming In 238
16.5 Panning or Scrolling 240
16.5.1 Example: Scrolling the Plotting Area 241
16.6 Selecting Title and Axes 242
16.7 Plotter Translations and Actions 243
16.7.1 Example Translation Tables 247

17. Defining Custom Markers 249

17.1 Creating a Custom Marker 250
17.2 PostScript Output of Custom Markers 252
17.3 Naming and Looking Up Markers 257

18. Defining Custom Fill Patterns 259

18.1 Creating a Custom Fill Pattern 260
18.1.1 Defining a Fill Pattern for X 260
18.1.2 Defining a Fill Pattern for PostScript 262
18.2 Naming and Looking up Fill Patterns 265

Appendix A Widget Resource Reference 267

A.1 Annotation Resources 268
A.2 Axis Resources 270
A.3 BarPlot Resources 272
A.4 CallbackPlot Resources 273
A.5 ErrorPlot Resources 274
A.6 FadePlot Resources 275
A.7 HighLowPlot Resources 276
A.8 HistoPlot Resources 277
A.9 LinePlot Resources 278
A.10 PiePlot Resources 279
A.11 Plotter Resources 280

Appendix B Function Reference 283

B.1 Annotation Functions 284
B.2 Axis Functions 285
B.3 BarPlot Functions 286
B.4 CallbackPlot Functions 286
B.5 ErrorPlot Functions 287
B.6 FadePlot Functions 287
B.7 FillPattern Functions 287
B.8 HighLowPlot Functions 288
B.9 HistoPlot Functions 289
B.10 LinePlot Functions 290
B.11 Marker Functions 291
B.12 PiePlot Functions 291
B.13 Plotter Functions 291

Appendix C Data Type Reference 293

C.1 Callback Structures 294
C.2 Callback Reasons 295
C.3 Data Types 295
C.4 Fill Patterns 296
C.5 Line Patterns 297
C.6 Markers 298

Index. 299

Figures

Page

Figure 1-1. The widget class hierarchy of the XiPlotter library 5
Figure 1-2. A typical widget instance hierarchy using the XiPlotter 6
Figure 3-1. Predefined markers 29
Figure 3-2. Predefined fill Patterns. 32
Figure 3-3. Predefined Line Patterns 34
Figure 3-4. Definitions of the standard line patterns 34
Figure 4-1. The XiPlotData structure and the XiPlotDataType enumerated type 40
Figure 4-2. Specifying plot data through XiPlotData resources 41
Figure 5-1. The XiPlotter legend 62
Figure 5-2. XiPlotterPrint() 73
Figure 6-1. Resources that control Axis appearance 83
Figure 6-2. Plotter widgets with logarithmic axes 96
Figure 6-3. XiAxisSetMarks() 99
Figure 6-4. XiAxisSetLabels() and XiAxisSetLabelStrings() 104
Figure 7-1. Functions for specifying (X,Y) LinePlot data in simple arrays. 118
Figure 7-2. Functions for specifying LinePlot data in structures 119
Figure 7-3. XiLinePlotDataChanged 120
Figure 7-4. XiLinePlotDataAdded() 121
Figure 8-1. ErrorPlot functions for specifying error values in simple arrays 130
Figure 8-2. XiErrorPlotSetErrors() 131
Figure 9-1. The XmNbarSize and XmNbarPosition resources 140
Figure 9-2. Independently positioned groups of bars 141
Figure 9-3. Simple functions for specifying BarPlot data 144
Figure 9-4. XiBarPlotSetValues() 144
Figure 10-1. Specifying HistoPlot data in simple arrays 151
Figure 10-2. XiHistoPlotSetValues() 152
Figure 12-1. Visual resources of the HighLowPlot widget 170
Figure 12-2. Specifying data as arrays of doubles, floats, or ints 173
Figure 12-3. XiHighLowPlotSetValues() 174
Figure 12-4. XiHighLowPlotSetPoints() 175
Figure 12-5. XiHighLowPlotDataChanged() 176
Figure 13-1. Positioning annotations 184
Figure 13-2. Annotation offsets 186

Examples

Page

Example 2-1. The XiPlotter Hello, World program 12
Example 3-1. Specifying text style an size with a single font list 25
Example 4-1. Specifying data in simple arrays 44
Example 4-2. Specifying data in arrays of structures 45
Example 4-3. Specifying plot data of mixed types 46
Example 4-4. Specifying linear data 47
Example 4-5. Performing a linear transform on data values 48
Example 5-1. Creating XiPlotter widgets 52
Example 6-1. Querying the Axes of a Plotter widget 79
Example 7-1. Two methods for creating a LinePlot widget. 110
Example 7-2. Using XiLinePlotDataAdded() 122
Example 8-1. Creating ErrorPlot widgets 126
Example 8-2. Specifying error values from structures 132
Example 9-1. Creating BarPlot Widgets 136
Example 10-1. Creating a HistoPlot widget 148
Example 11-1. Creating a piechart with PiePot 157
Example 12-1. Creating a HighLowPlot widget 168
Example 13-1. Creating annotations 181
Example 13-2. Thoroughly annotating a PiePlot wedge 199
Example 14-1. Using the FadePlot widget 205
Example 14-2. FadePlot resources in a resource file 206
Example 14-3. Example non-linear interpolation functions 207
Example 15-1. Using the CallbackPlot widget 221
Example 15-2. Resources for the CallbackPlot Example 225
Example 16-1. Highlighting a selected plot 230
Example 16-2. Hit detection 232
Example 16-3. Displaying Mouse Coordinates 236
Example 16-4. Zooming In 239
Example 16-5. Scrolling the plotting area 242
Example 16-6. Example Translation Tables 247
Example 17-1. Creating a custom marker (X only) 252
Example 17-2. PostScript code to draw the marker 254
Example 17-3. Defining and using custom markers for X and PostScript 256

Example 18-1. Creating an XiFillPattern (X only) 261
Example 18-2. Creating an XiFillPattern for X and PostScript 265

Tables

Page

Table 3-1. Predefined Markers. 30
Table 3-2. Predefined fill Patterns 31
Table 5-1. Plotter Resources 53
Table 6-1. Axis Resources 80
Table 7-1 LinePlot Resources 111
Table 8-1. ErrorPlot Resources 127
Table 9-1. BarPlot Resources. 138
Table 10-1. HistoPlot Resources 149
Table 11-1. PiePlot Resources 158
Table 12-1. HighLowPlot Resources 169
Table 13-1. AnnotationPlot Resources 182
Table 14-1. FadePlot Resources. 202
Table 15-1. CallbackPlot Resources 212

Preface

The XiPlotter library is a set of X Toolkit widgets and their related functions for displaying (on an X display) and printing (to a PostScript printer) line plots, line plots with error bars, bar plots, histograms, pie charts, and high-low-close plots. This manual documents the XiPlotter library.

About the Software

The XiPlotter library requires Motif 1.2 and the X11R5 Xlib and Xt libraries. The Plotter widget is subclassed from the Motif XmBulletinBoard widget.

The X protocol has been stable since X11R3, so these widgets will display graphs correctly with any X server from X11R3 on. The widgets can print themselves on any PostScript printer.

The Plotter library is implemented as a single Plotter widget that provides a plotting infrastructure, and a number of separate Plot widgets that display line plots, bar plots, pie charts and so on. These Plot widgets are bundled with the Plotter library. This manual describes the complete library, with each of the Plot widgets.

About This Book

This manual is the complete documentation for the XiPlotter library. It is organized in "programmer's guide" form, rather than as a reference manual, which means that you should read it more-or-less from beginning to end (with exceptions as noted below), in order to learn about each of the important features of the Plotter library. Once you have read about all of these features, you'll find the quick-reference section at the end of the manual quite useful during your day-to-day programming.

The book has the following structure:

Chapter 1: *Introduction: Plotter Architecture*, is an introduction to the widgets and other components of the Plotter library, and an explanation of how they fit together.

Chapter 2: *“Hello, World”*, presents a complete (though simple) working example of using the Plotter library to display plots. It contains information on compiling and linking applications that use the Plotter library, so you can get up and running right away.

Chapter 3: *Plotter Data Types and Resources*, explains some special data types that are commonly used in the Plotter library, and also provides hints about specifying values for other common resource types. The material in this chapter is a little dry, but it will pay to be familiar with it.

Chapter 4: *Plot Data Representation*, documents the XiPlotData abstraction; this is the abstraction used by almost all of the widgets in the library for data specification. Using XiPlotData values, you can very flexibly instruct your plots to extract data from arrays of arbitrary application data structures.

Chapter 5: *The XiPlotter Widget*, documents each of the resources and constraint resources of the XiPlotter widget, as well as its convenience functions, callbacks, translations, and actions. This chapter also explains how you can generate a PostScript representation of your plots.

Chapter 6: *The XiAxis Widget*, explains all of the XiAxis resources, and documents a number of important functions for configuring the X and Y axes of the Plotter.

Once you have read these first six chapters, you will understand the fundamental architecture of the XiPlotter library. The next six chapters document the individual Plot widgets responsible for displaying different kinds of graphs. You can read them in any order you prefer, and need only read the chapters for the widgets you plan to use.

Chapter 7: *The XiLinePlot Widget*, explains how to display line plots.

Chapter 8: *The XiErrorPlot Widget*, explains how to display line plots with error bars.

Chapter 9: *The XiBarPlot Widget*, explains how to display bar charts and stacked bar charts.

Chapter 10: *The XiHistoPlot Widget*, explains how to displayhistograms.

Chapter 11: *The XiPiePlot Widget*, explains how to display piecharts.

Chapter 12: *The XiHighLowPlot Widget*, explains how to display high-low-close plots of financial market or similar data.

Chapter 13: *Annotating Plots*, explains how you can add textual and graphical annotations to the Plotter widget.

Chapter 14: *The FadePlot Widget*, documents a special Plot widget that provides a smoothly shaded background for presentation graphics.

Chapter 15: *Drawing Your Own: the CallbackPlot*, explains how you can do your own custom drawing into the Plotter widget.

Chapter 16: *User Interaction with the Plotter*, explains the callbacks the Plotter provides for handling user interaction with the Plotter, and also the actions and translations that invoke those callbacks.

The final two chapters cover advanced topics for customizing the Plotter. You need not read them your first Time through the manual.

Chapter 17: *Defining Custom Markers*, explains how to define custom line plot marker glyphs for use in X and PostScript.

Chapter 18: *Defining Custom Fill Patterns*, explains how to define custom hatch patterns for use in X and PostScript.

The last part of the manual is a quick reference section that you may want to refer to during your day-to-day use of the Plotter library.

Appendix A: *Widget Resource Reference*, contains tables of resources for each widget in the library. The tables are cross-referenced by page number back to the main body of the book, so you can easily look up a full explanation of any item.

Appendix B: *Function Reference*, lists the signatures of every public function defined by the Plotter library. The list is organized by widget or functional group.

Appendix C: *Data Type Reference*, shows the definitions of all the public data types used by the Plotter library, and also contains tables of predefined markers, fill patterns, and line patterns.

Assumptions and Recommended Reading

In order to use the Plotter library, you will need to be comfortable with the C programming language and the X Toolkit. Throughout this manual, I assume that you are at least familiar with the basic X Toolkit programming model--that you understand what a widget is, and know the difference between a widget instance and a widget class, for example.You will also need to understand widget resources and how to set them, and callback lists and how to specify them.

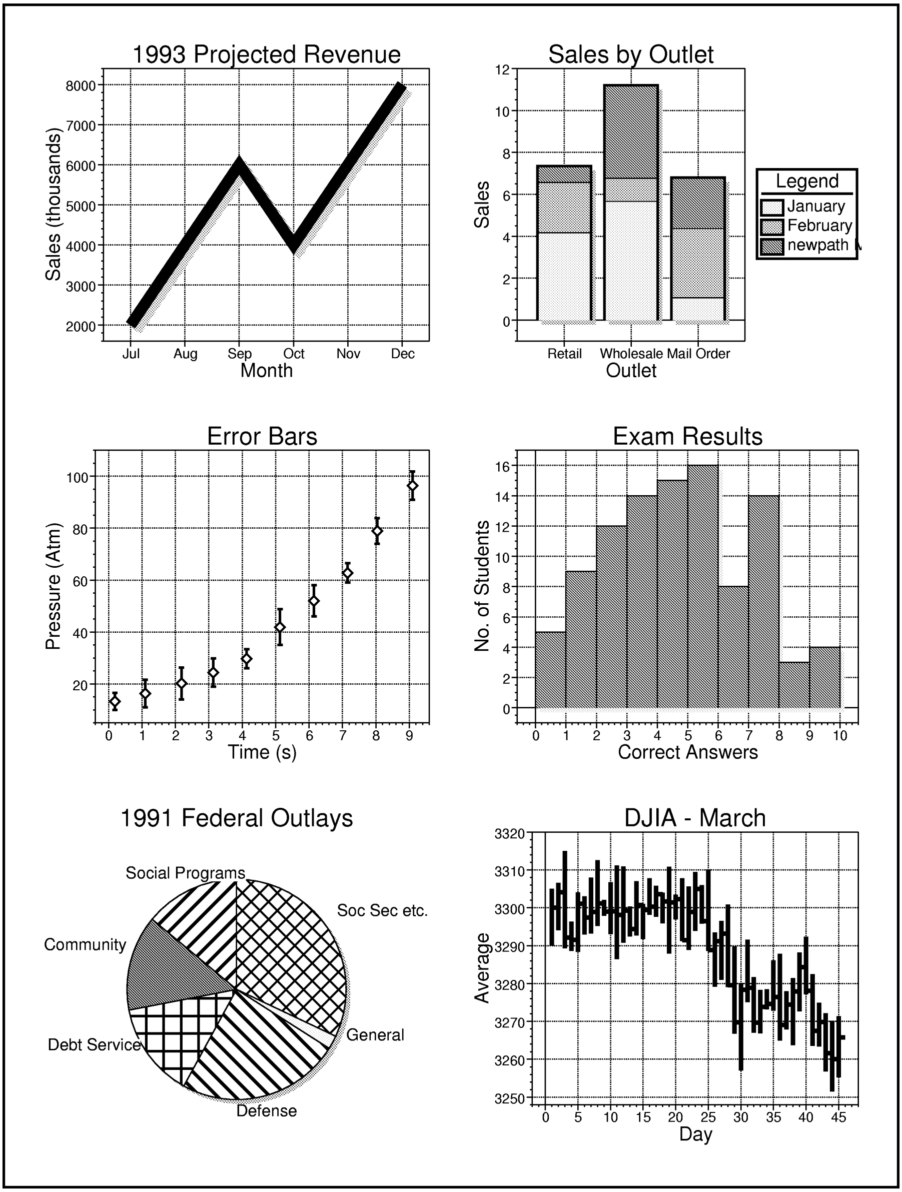
O'Reilly & Associates publishes a useful series of books on programming with X. If you are new to the X Toolkit, you may be interested in Volume Four-*Toolkit Intrinsics Programming Manual*. This volume provides complete documentation for programming with Xt, and does not omit the advanced topics as many introductory books do. Whether or not you are new to Xt programming, I recommend Volume Five-*Toolkit Intrinsics Reference Manual*; it is a complete and handy reference to Xt, with far more explanation of functions than you will find in the standard Xt man pages.

Motif widget set

The Plotter library requires the Motif widget set. O'Reilly's VolumeSix-*Motif Programming Manual* (a.k.a "the blue book") may be useful to you. Volume 6B-*Motif Reference Manual* is also useful; it is a complete man page reference for Motif 1.2. I find it easier to use than the standard OSF reference.

Although the X Toolkit is fairly effective at hiding many of the details of Xlib from the programmer, displaying graphs is a task that relies heavily on Xlib drawing commands. Because the visual appearance of the plots is customizable, there are a number of widget resources, such as colors, line widths, and line patterns, that are Xlib data types. Chapter 3, *Plotter Data Types and Resources* resources explains what you need to know about these data types, but a basic knowledge of Xlib will be useful. Volume One in the O'Reilly series, *Xlib Programming Manual*, is a definitive book on all aspects of Xlib programming. The accompanying reference is Volume Two-*Xlib Reference Manual*.

Finally, if you will be defining custom markers or fill patterns that will be printed as well as displayed in X, (there are plenty of predefined markers and fill patterns, so this is not something that you will have to do) you will have to write some simple PostScript code--you will need to understand the PostScript programming model, and be familiar with the basic PostScript drawing operators. The book that defines the PostScript language is *PostScript Language Reference Manual*, Second Edition, published by Addison-Wesley. This large book contains a detailed introduction to the PostScript programming model, and a complete language reference section.


1993 Projected Revenue
Sales (thousands)
8000
7000
6000
5000
4000
3000
2000
Jul
Aug
Sep
Oct
Nov
Dec
Month
Sales by Outlet
Sales
12
10
8
6
4
2
0
Retail
Wholesale
Mail Order
Outlet
Legend
January
February
newpath
Error Bars
Pressure (Atm)
100
80
60
40
20
0
1
2
3
4
5
6
7
8
9
Time (s)
Exam Results
No. of Students
16
14
12
10
8
6
4
2
0
0
1
2
3
4
5
6
7
8
9
10
Correct Answers
1991 Federal Outlays
Social Programs
Soc Sec etc.
Community
General
Debt Service
Defense
DJIA - March
Average
3320
3310
3300
3290
3280
3270
3260
3250
0
5
10
15
20
25
30
35
40
45
Day

1

Introduction: XiPlotter Architecture

The XiPlotter library provides X Toolkit widgets and related convenience functions for displaying data with six standard plot types: line plots (sometimes known as x-y plots), error plots, bar plots, histograms, pie charts, and high-low-close plots (for financial market data, for example). The facing page shows some typical plots that can be generated with this library. (The pictures of example plots throughout this book were all generated using the PostScript output capability of the Plotter library itself).

The features of the Plotter library include the following:

- All plot types can be displayed in X, or output as high-resolution encapsulated color PostScript for printing or inclusion into other documents. (This is native PostScript output, not simply a window dump.) The widgets support internationalized text.
- The Plotter widget itself has 87 resources that control the general appearance and behavior of the Plotter title, legend and plotting area. Each of the three possible axes (the X axis and the primary and secondary Y axes) is implemented as a separate object with 65 of its own resources to control its appearance and numbering. Finally, the various supported plot types add another 220 resources. If we count the Axis resources three times, for the three independent axes, this is over 500 resources in the Plotter library.
- The X and Y axes support auto-scaling, auto-labeling, and auto-numbering. Axis labels may be numbers or strings. Logarithmic axes are supported and plot data will be automatically scaled to a logarithmic axis. An optional "secondary Y axis" can be displayed to the right of the plotting area.
- The Plotter widget is implemented as a "constraint widget", and each of the plot types are implemented as a separate gadget class to be created as children of a Plotter. This modular architecture makes it trivial to display multiple plots and plots of multiple types within a single widget.
- The Plotter widget displays an optional legend. Legend entries for plots are specified through a constraint resource.

- The Legend is sensitive to mouse clicks, and behaves like a List widget, allowing users to select individual plots within the widget. The plotting area of the widget is also sensitive to mouse clicks, and drags and can display crosshairs and allow the user to select a region by dragging out a rectangle with the mouse.
- Data is specified for all plot types with a very flexible scheme that allows the plot widget to extract values of any floating-point or scalar type directly from application data structures--there is no need to make a special copy of the data just for the plot widget.
- The Plotter library provides 12 standard marker types, each in four different sizes, to mark data points in line plots and error bar plots. There are also 9 predefined dash patterns for the lines that connect the data points. For bar plots and pie charts, there are 25 predefined patterns for filling the bars and pie slices. You can also create your own custom markers, line patterns, and fill patterns.

1.1 XiPlotter Architecture[1]

The XiPlotter widget is a subclass of the Motif XmBulletinBoard widget class. It displays a title and a legend and provides a general widget plotting infrastructure (such as handling autoscaling and generating mouse selection callbacks), but does not itself display any data. That is the job of separate Plot objects, implemented as independent widget classes, and instantiated as children of the Plotter widget.

These children of the Plotter widget are all subclasses of the XiPlot meta-class, which is itself a subclass of the Xt RectObj class. There are six of these Plot widget classes currently implemented, each of which displays a different kind of plot: XiLinePlot, XiErrorPlot, XiBarPlot, XiHistoPlot, XiPiePlot, and XiHighLowPlot. The XiAxis widget is also a subclass of the XiPlot meta-class; the XiPlotter uses this widget class internally to display its axes. Figure 1-1 shows the widget class hierarchy of the XiPlotter library.

Like all widgets, each of the Plot widget classes has its own set of resources that control the appearance and behavior of any particular instance of that widget class. Each of the Plot widget classes provides resources that you can set to specify the data to be plotted, and the graphical appearance (colors, line widths, etc.) of that data.

[1] If you are impatient with manual reading, you may want to skip to Chapter 2, *Hello World* at this point. Be sure to return to this chapter, however; it provides an important overview of the library.

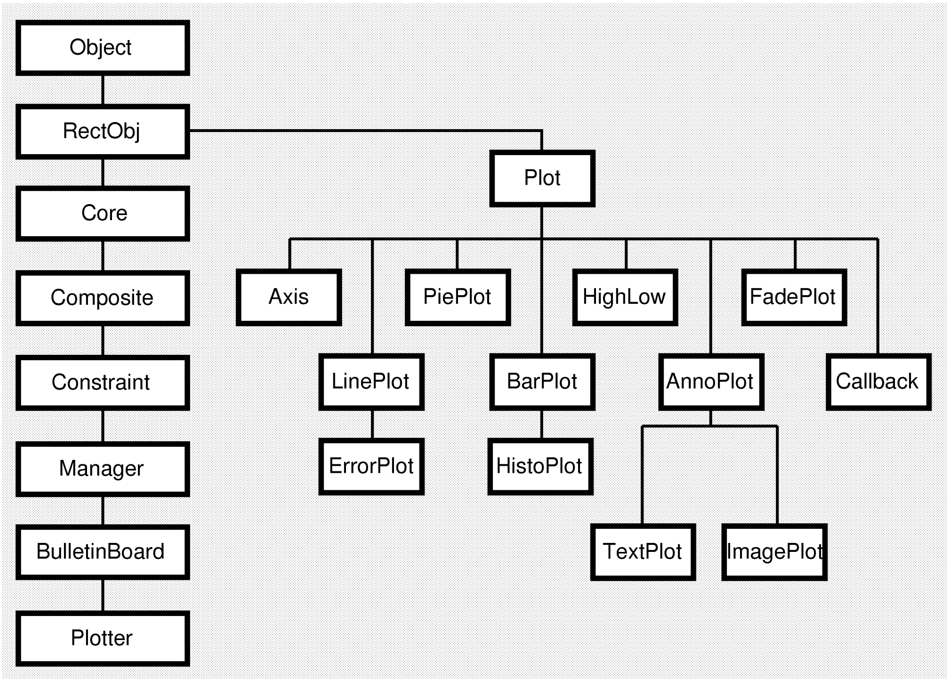


Figure 1-1. The widget class hierarchy of the XiPlotter library

Also like all widgets, each Plot widget has various "class methods" that it implements. The XiPlot widget class requires that each of its subclasses implement methods for displaying their data in X, for printing their data in PostScript, for rescaling their data when the bounds of the axes change or when the size of the Plotter widget changes, and so on. This set of class methods is essentially the definition of what it means to be a Plot widget, and is the interface definition for interaction with the Plotter widget.[1]

The Plotter library contains six standard plot types that implement each of these required methods, but any subclass of XiPlot that implements the correct methods will be a legal Plot type, suitable for use with the XiPlotter widget. (Writing widgets is a tricky business, well beyond the scope of this manual, but if you are an experi-

[1] Don’t worry if you do not fully understand this paragraph. You can use the XiPlotter library and its widgets without knowing the details of their implementation.

enced widget writer, and have purchased the source code to the XiPlotter library, you will be able to create new plot types beyond the six basic ones supplied with the library.)

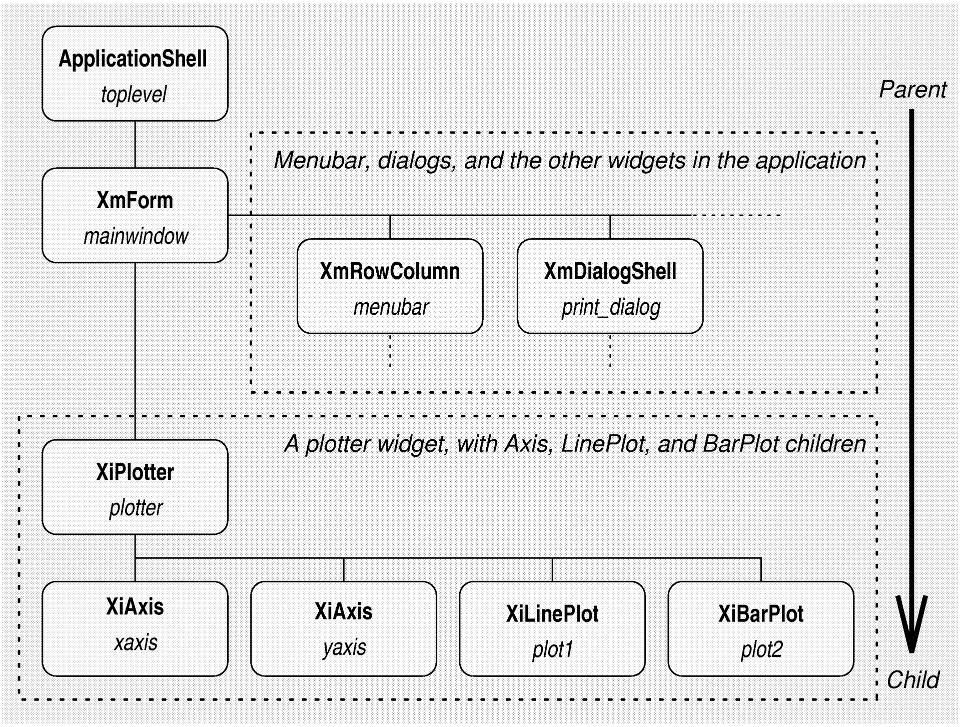


Figure 1-2. A typical widget instance hierarchy using the XiPlotter

Because the Axis and the various plot widgets are sub-classes (sub-sub-classes, really) of the Xt RectObj class, they do not have windows of their own. This kind of windowless widget is sometimes called a "gadget". (But don't confuse this with the Motif XmGadget class--Motif Gadgets are a specific kind of the more general category of gadgets.) In order for a gadget to correctly display itself, it must have a cooperating parent widget into whose window it can draw. The XiPlotter widget serves this purpose for all of the gadgets in the library--Plots can only be instantiated as direct children of a Plotter widget. Figure 1-2 shows the widget *instance* hierarchy of a typical XiPlotter application.

Note that two Axis widgets appear in the figure as children of the Plotter. These widgets display the X and Y axes, and are automatically created and managed by the Plotter widget. The Axis functionality could have been incorporated directly into the Plotter widget, but having independent Axis objects is useful because they serve as important placeholders for axis resources--rather than adding X and Y axis resources to the Plotter widget, these resources are broken into two independent sets of Axis widget resources.

Keeping Plotter, Axis, and Plot widgets separate has several advantages:

- Plots can be created, managed, unmanaged, and destroyed in exactly the same way you would handle any other widget.
- Creating multiple plots in a Plotter widget is simply a matter of creating multiple Plot objects. Since plots are independent objects you can always display multiple kinds of Plot objects (two bar charts and a line plot, for example) in a single Plotter widget.
- Since each plot is a separate object, each has its own set of resources, and there is no need to specify colors, line patterns, fill patterns, and so on in cumbersome arrays.
- Since the Plotter is a Constraint widget, it provides constraint resources that each of its children can set independently. It uses constraints to specify the name to appear in the legend for each plot, to specify whether a plot is selected or not, and to specify the stacking order of the plots (which affects how multiple plots overlap each other.)
- It is possible (with source code and appropriate expertise) to create new Plot types to work with the Plotter widget.

1.2 XiPlotter Data Types

One of the primary features and design goals of the Plotter library was that it should be able to display plots both in an X window, and equally well on a printed page through PostScript. In order to achieve this, the Plotter library implements a number of data types that hide the (significant) differences between the X and PostScript display models. A number of these abstractions are internal to the library, but there are two that you will encounter while using the library:

XiMarker

Line plots are often drawn with a small glyph or "marker" displayed at each data point. If plots were only to be displayed in X, it would be sufficient to simply specify Xlib Pixmap to use as a marker, but for use with PostScript, an XiMarker is required.

XiFillPattern

Bar charts and pie charts often use "hatch pattern" or "fill patterns" to distinguish multiple bars or pie wedges from each other. As with line plot markers, a Pixmap would be sufficient if the plots were to be displayed in X alone, but to support PostScript, the Plotter library uses the XiFillPattern abstraction.

The Plotter library provides a number of pre-defined markers and fill patterns--you can use one of these predefined values any time you need to specify a marker or pattern. Chapter 3, *Plotter Data Types and Resources* explains these types in more detail and lists their predefined values. It is also possible to define your own custom markers and fill patterns. Doing so is explained in Chapter 17 and Chapter 18.

1.3 Plot Data Representation

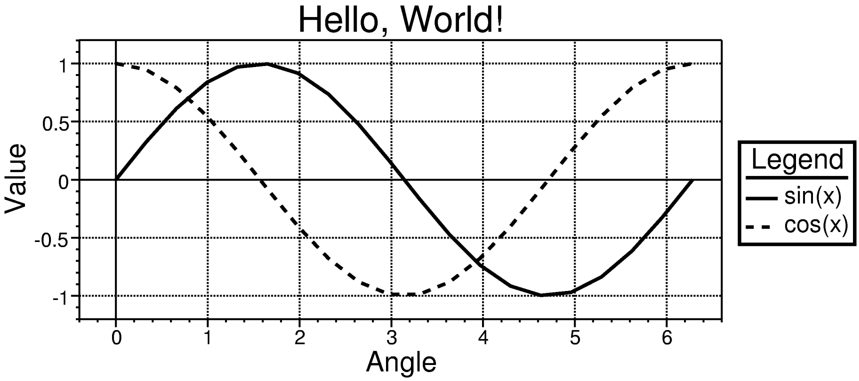
Most applications do not simply plot static arrays of data. First they will obtain the data by reading a file, querying a database, or performing a numeric integration, for example. Besides plotting the data, they might display it in a table, perform statistical analyses on it, or write it out in a new file format. Because different applications have different goals, each may choose to store its data in different structures. One application might store (x,y) line plot values in two separate arrays of double, for example, and another might store line plot data as integers in an array of structures, each structure with an x and a y field.

A fundamental architectural goal of the Plotter library is that applications should not have to make special copies of their data just for their Plot widgets, and should not have to make their data structures conform to whatever those Plot widgets use internally. So instead of specifying a required format for plot data, the various Plot widgets have resources that let you describe the format of the data you will supply.

Data is assumed only to be in an array of structures--quite a broad assumption--and you can specify the size of each element of the array, the type of the data value stored within each element (the widgets understand nine floating point and scalar types) and the offset of the data value from the beginning of the structure. You also, of course, provide a pointer to the array and specify the number of elements in the array. With this size, type, and offset information, the Plot widgets can extract and plot the data values they need from whatever application data structures you find it convenient to use. The default values of the size, type, and offset resources are chosen so that if you store your data in arrays of double, you will not need to set any of the resources. (In fact, you often needn't use widget resources for data specification at all--the Plot widgets each provide convenience functions that make it easier to specify these values.)

One other feature of this data representation model is that when your data is linearly spaced (many line plots, for example, display sampled data that is uniformly spaced along the X axis) you need not specify an array of data at all; instead, you can simply specify the minimum data value, the number of values, and the interval between values. You can use a similar technique to apply an arbitrary linear transformation to any array of data.

This data representation (and transformation) scheme is described in detail in Chapter 4, *Plot Data Representation.*


Hello, World!
Value
1
0.5
0
-0.5
-1
0
1
2
3
4
5
6
Angle
Legend
sin(x)
cos(x)

2

Hello, World

This chapter presents and explains a complete (if somewhat simple) working Xt program that uses the XiPlotter library to display plots. It also explains how to compile the program and link it with the XiPlotter library. The figure on the facing page is the PostScript output of the program.

2.1 The Program

Example 2-1 shows a complete Xt program using the XiPlotter library.

This program breaks down into a few major sections, each with some notable features:

Header Files (lines 1-4)

> <*Xm/Xm.h*> is the standard Motif header file, and you should include it in all your Motif programs. <*Xi/Plotter.h*> is the main header file for the Plotter library; you must include it in all your Plotter applications. <*Xi/LinePlot.h*> is the standard public header file for the LinePlot widget; you must include it in any application that uses the that widget. Similarly, you would include <*Xi/BarPlot.h*> and <*Xi/PiePlot.h*> in applications that used the BarPlot and PiePlot widgets. We also include the standard header <*math.h*> because we will be doing some trigonometry in the following code.

Xt Housekeeping (lines 11-16, 22-24, 105, 116)

> These lines are standard parts of any Xt application. They initialize the toolkit, "realize" all the widgets, and enter the main event loop.

Example 2-1. The XiPlotter Hello, World program

```
1 #include <Xm/Xm.h>
2 #include <Xi/Plotter.h>
3 #include <Xi/LinePlot.h>
4 #include <math.h>
5
6 typedef struct { /* the data type we'll use to store data */
7float x;
8float y;
9 }Point;
10
11 void main(ac, av)
12 int ac;
13 char *av[];
14 {
15XtAppContext app;
16Widget toplevel, plotter, sine_plot, cosine_plot;
17int i;
18Point sine_data[20];
19float cosine_data[20];
20double x, dx;
21
22/* initialize the X Toolkit as usual */
23toplevel = XtVaAppInitialize(&app, "hello", NULL, 0,
24&ac, av, NULL, NULL);
25
26/* create the plotter widget as a child of the top level shell */
27plotter = XtVaCreateWidget("plotter", xiPlotterWidgetClass,
28                                toplevel,
29                                XmNwidth, 600,
                               XmNheight, 250,
                               XmNtitle, "Hello, World!",
                               NULL);
    /* and manage it */
    XtManageChild(plotter);

    /*
     * Specify titles for each axis.
     * by default, the axes are autoscaled, auto-numbered, and
    * auto-labeled, so there are no other resources we must set.
     * Note how we get the Axis objects from the Plotter widget.
     */
  XtVaSetValues(XiPlotterXAxis(plotter),XmNtitle,"Angle", NULL);
  XtVaSetValues(XiPlotterYAxis(plotter),XmNtitle,"Value", NULL);

    /*
     * Compute some data for the line plots.
     * In an application, this data could come from any source.
     * For demo purposes, we store the values in two ways.
     */
    dx = 2*3.1415926/(float)(XtNumber(sine_data)-1);
    for (i=0; i < XtNumber(sine_data); i++) {
        extern double sin(), cos();
        x = i*dx;
        sine_data[i].x = x;
        sine_data[i].y = sin(x);
        cosine_data[i] = cos(x);
    }

    /*
     * Create two line plot widgets as children of the Plotter,
     * and set various resources on them. Notice that we do
     * not specify any data here.
     */
    sine_plot = XtVaCreateWidget(
                    "sine", xiLinePlotWidgetClass, plotter,
                    XmNlegendName, "sin(x)",
                    XmNlineWidth, 2,
                    XmNconnectPoints, True,
                    XmNmarkPoints, False,
                   XtVaTypedArg,XmNlineColor,XmRString,"navy",5,
                    NULL);
    cosine_plot = XtVaCreateWidget(
                    "cosine", xiLinePlotWidgetClass, plotter,
                    XmNlegendName, "cos(x)",
                    XmNlineWidth, 2,
                    XmNlinePattern, XiLinePatternDashed2,
                    NULL);

    /*
     * Specify data for the 1st line plot by specifying the type
    * and position of the data fields within the Point structure
     * we used. We could also do this with resources.
     */
    XiLinePlotSetPoints(sine_plot,
                         sine_data, XiPlotDataFloat,
                         sizeof(Point), XtOffsetOf(Point, x),
                         sine_data, XiPlotDataFloat,
 88                              sizeof(Point), XtOffsetOf(Point, y),
 89                              XtNumber(sine_data));
 90
 91     /*
 92      * Specify data for the next line plot with a different,
 93      * simpler convenience function. Here we take advantage that
 94      * the X values are linear, & the Y values in a simple array.
 95      */
 96     XiLinePlotSetFloatYValues(cosine_plot,
 97                                  0.0, dx,  /* x min, x increment */
 98                                cosine_data, XtNumber (cosine_data));
 99
100     /* don't forget to manage the plots, or they won't appear! */
101     XtManageChild(sine_plot);
102     XtManageChild(cosine_plot);
103
104     /* Standard Xt: realize the shell */
105     XtRealizeWidget(toplevel);
106
107     /*
108      * Print the plot to the file hello.ps, at 5 x 3 inches,
109      * using a scale factor of 1.0, and not using landscape mode.
110      * Normally, we'd print the plot under user control, but this
111      * is here for demonstration purposes.
112      */
113     XiPlotterPrint(plotter, "hello.ps", 5*72, 3*72, 1.0, False);
114
115     /* last, enter the standard Xt event loop. */
116     XtAppMainLoop(app);
117 }
```

Creating the Plotter (lines 26-34)

These lines create and manage the Plotter widget, setting some Plotter resources at the same time. We could also have used XtCreateWidget(), XtCreateManagedWidget(), XtVaCreateManagedWidget(), or XiCreatePlotter() to do this. Chapter 3, *The Plotter Widget* documents the Plotter widget and all of its resources.

Configuring the Axes (lines 42-43)

These lines configure the Axis widgets. The X and Y Axis widgets are automatically created by the Plotter widget; you may often want to set resources on these widgets before displaying any plots. In this example, we simply set the XmNtitle resource on each widget--the Axes will, by default, automatically choose appropriate bounds, tick mark positions and tick mark labels, so there is nothing else that we need to configure. We set the resources with the standard XtVaSetValues() function, but note that we have to get the widget pointers for the X and Y axes from the Plotter widget by calling the functions XiPlotterXAxis() and XiPlotterYAxis(). Chapter 6, *The Axis Widget* documents the Axis widget and all of its resources.

Computing Data (lines 50-57)

This code does some trigonometry to compute the data points we will be plotting. Some applications will get their plot data in this way, but many will instead read it from data files or query it from a database. Notice that we store X and Y values for the sine plot, but only Y values for the cosine plot. This is simply so we can demonstrate two methods of specifying the data for the plots.

Creating Plots (lines 64-78)

These two function calls create two LinePlot children of the Plotter widget. As explained in the previous chapter, plots are added to a Plotter widget in the same way that any widget is created. They are also managed, unmanaged, and destroyed in the same way as any other widget. We set a few resources on each widget that will control its appearance, but we do not specify any data values here. (All the LinePlot resources are documented in Chapter 7, *The LinePlot Widget* ; see that chapter if you want to look these resources up.) Note that we set the XmNlegendName resource for both LinePlot widgets. This is actually a constraint resource provided by the Plotter widget; it is used to specify the string that should appear in the Plotter legend. Note also that we do not manage the widgets here. We'll manage them after we specify data for them. (In this example, it doesn't matter much when we manage these LinePlot widgets, because the Plotter widget has not been realized yet, but if we were adding these plots dynamically to an existing widget, it could be significantly more efficient to specify the data before managing the widgets.)

Specifying Data (lines 84-99)

The code on these lines call LinePlot convenience functions to tell the widgets what data they should display. The second function, XiLinePlotSetFloatYValues() is the simpler one--since our X values are evenly spaced, we simply specify a minimum X value, an X increment, and an array of float to plot as the Y values for each point. We could have used this same technique for the first plot as well, but for this demonstration, we used a more complicated function. XiLinePlotSetPoints() tells the LinePlot widget how to extract its X values and its Y values from our Point data structure. For both the X and the Y values, the arguments to this function specify a pointer to an array of the Point data type we defined, the type of the value to extract from each element of this array (a float), the size of the Point data structure, and the offset of the value within the structure. Since the X and Y values are stored in the same array, with the same data type, only the X and Y offset arguments are different. Note that these arguments are computed with XtOffsetOf(). This is a standard Xt macro, but unless you have written widget, or used application resources, you may not be familiar with it. We could also have specified data for the plots with LinePlot resources similar to the arguments to these convenience functions. Chapter 4, *Plot Data Representation* explains in detail how to specify plot data with resources or with convenience functions.

Making the Plots Visible (lines 101-102)

Now that the plots have had data specified, we manage them, exactly as any other widget or gadget would be managed.

Printing the Plotter (line 113)

This line calls XiPlotterPrint() to produce a PostScript representation of the Plotter and its LinePlot widgets. The width and height arguments to this function are measured in PostScript points, so we multiply inches by 72 to convert. These arguments are optional; if omitted, the function will use the largest size that will fit on the page that is consistent with the Plotter's on-screen aspect ratio. Chapter 5, *The Plotter Widget* plotter explains this printing function in more detail.

2.2 Compiling and Linking the Program

You can compile a program using the XiPlotter widget much as you would compile any other Motif application. The only thing you must be careful of is to ensure that the compiler can find the header files for the EPak widgets--use the CW-I option for this. So, if you had the EPak header files installed in/ *usr/local/include/Xi*, and had your X11 and Motif header files installed in the default */usr/include/X11* and */usr/include/Xm* directories, then you could compile our hello.c program to a hello.o object file with a command like the following:

```
cc -c -I/usr/local/include hello.c
```

The EPak header files have #ifdef'ed function declarations for both old K&R-style C compiler, and newer ANSI-C compilers. If you are using X11R5, you will automatically get the ANSI-C function prototypes when you use an ANSI-C compiler that defines the__STDC__ symbol. On some systems, you may need to add -DXT-FUNCPROTO to the compiler command line in order to get these ANSI-C headers.

Linking our "hello, world" program is similarly simple. You must use the -L flag to tell the compiler where to find the Plotter library. The name of the library is *libBPak.a*, and you can link with it with -lBPak. Assuming you had installed the library as */usr/local/lib/libBPak.a*, and assuming that you have your Motif, Xt and X libraries installed in the default */usr/lib/*, then you could link hello.o with a command like the following:

```
cc -o hello hello.o -L/usr/local/lib -lBPak -lXm -lXt -lX11 -lXext -lm
```

Note that we used -lm on the command line to link with the math library. The *hello.c* program used the sin() and cos() functions from this library, and the Plotter library itself uses math functions as well (mostly in the code for supporting logarithmic axes.) What this means is that you must always link with the math library when using the Plotter library.

Also note the use of -lXext. This links the application with the X extensions library. This is necessary, because the Plotter widget uses the MBX extension to the X protocol, if it is supported by the X server, to implement double buffering.

These examples use cc to invoke the C compiler. The preferred compiler might be something different on your system--gcc or acc, for example. Also, you might have to add command line flags specific to your system and compiler.

3

Plotter Data Types and Resources

Like all widgets, the Plotter widget, the Axis widget, and the various Plot widgets that accompany them are controlled via a set of named resources. You can specify the appearance and behavior of these widgets by providing values for these resources with the functions XtSetValues() and XtVaSetValues().

What is different about the widgets in the Plotter library is the types of some of their resources. While many widgets use only resources that are strings or scalar types, the Plotter widgets must also use floating point numbers, and a number of special types, like XiFillPattern that are unique to this library. This chapter explains all you need to know to use these data types and to use resources of these types. Some of this will be review for experienced Xt programmers, but we recommend that you read the entire chapter--there are some specialized tips here that even experienced programmers may find useful.

3.1 Double Resources

The task of plotting data inherently involves using floating points numbers, and the Plotter, Axis and Plot widgets have a number of resources of type double. One of the weak points of the X Toolkit architecture is that it is not good about handling resources of type double (or even of type float), and some care is required to set these floating point resources.

When you specify resource values for a widget using XtCreateWidget() or XtSetValues(), Xt expects each resource (name, value) pair to be stored in an Arg, shown below in an excerpt from *<X11/Intrinsic.h>*. This Arg structure contains an XtArgVal which, on most architectures, is 4 bytes--not large enough to contain a double.

```
typedef long XtArgVal;
typedef struct {
    String      name;
    XtArgVal    value;
} Arg, *ArgList;
```

To set a resource that is larger than an XtArgVal, Xt requires you to specify the address of the resource value rather than the value itself. This means that you cannot use floating-point constants in your code; you must store a constant value into a variable and then pass the address of the variable:

```
Widget xaxis, yaxis;
double min, max;
Arg args[10];
Cardinal i;
min = -10.0;
max = 10.0;
i = 0;
XtSetArg(args[i], XmNmin, &min); i++;
XtSetArg(args[i], XmNmax, &max); i++;
XtSetValues(xaxis, args, i);
```

The same holds when you use XtVaSetValues():

```
Widget xaxis, yaxis;
double min, max;
min = -10.0;
max = 10.0;
XtVaSetValues(xaxis,
              XmNmin, &min,
              XmNmax, &max,
              NULL);
```

You must be especially careful when using XtVaSetValues(), however, because the compiler cannot do type-checking on a variable-length argument list, and will allow you to pass a double instead of a double* without warning. Passing a double where a double* is expected places eight bytes on the stack instead of four. The first four will be interpreted as a pointer to the resource value, and the next four will be interpreted as the name of the next resource. Somewhere along the line this is quite likely to cause a core dump.

There is one more technique for setting double resources that is worth mentioning. This technique is a little inefficient, but does not require you to declare a dummy variable for each resource you want to set. XtVaSetValues() supports a special argument, XtVaTypedArg , which allows you to specify resource values as strings (or other types as well) and have Xt convert them to the appropriate type for you. Since pointers to string constants fit in four bytes, you can use a string constant instead of a double constant without the need for an intermediate variable. The code looks like this:

```
XtVaSetValues(xaxis,
XtVaTypedArg, XmNmin, XmRString, "-10.0", 6,
XtVaTypedArg, XmNmax, XmRString, "10.0", 5,
NULL);
```

When you use XtVaTypedArg in place of a resource name, the following 4 arguments specify the resource name, and the type, value, and size of the resource value. When using strings, the size should be the character length of the string as it would be returned by strlen(), plus one for the terminating null character.[1]

3.2 String And XmString Resources

Many of the resources of the Plotter and its associated widgets are strings or compound Strings. String and XmString resources are common in many widgets, and always require some special care, because conventions for handling them differ from widget to widget.

When you specify the value for any String or XmString resource in any of the Plotter library widgets, the widget will make a private, internal copy of the string or XmString. It does this so that it does not have to rely on your application to keep a copy of the string around. Often you will set string resources with string constants, which stick around anyway, but there are times when you might set these resources using allocated memory that will later be freed, or using a temporary buffer that will later have its contents changed.

If you query the value (using XtGetValues() or XtVaGetValues()) of any String resource of any Plotter library widget, the value that is returned to you is a pointer to the internal copy of the string. This copy is owned by the widget, not the application, and you must treat it as read-only: you are not allowed to modify or free the string. You should be aware that widgets are allowed, in general, to modify or free their strings at any time, so you should not rely on a pointer obtained by querying a

[1] In some versions of X11R6, using this XtVaTypedArg technique with double resources and XtVaSetValues() can cause a core dump. If this is the case on your system, you will have to revert to one of the other techniques described. Arg in calls to XtVaCreateWidget(), however.

resource to remain valid. If you need to use the value of a String resource that you have queried outside the scope of the procedure that queried the resource, you should make a private copy of the string for the application. (Use XtNewString() to allocate memory and copy the string, and remember to free the string with XtFree() when you are done)

3.3 Fonts and Displayed Text

The Plotter widget and its Axis, TextPlot and PiePlot children display text in a number of ways--the Plotter title, the axis labels, and the legend items, for example. All text is displayed in the same way that it is for the Motif XmLabel widget: an XmString is drawn, using the font or fonts specified in an XmFontList. Since XmStrings can contain any number of lines and any number of fonts, the text you display in the Plotter can be as complex as you want. The use of XmStrings in the Plotter also allows you to display internationalized text.

Wherever text is to be displayed in the Plotter, or in one of its Plot subclasses, you'll find an XmString resource that specifies the text to be displayed, and an XmFontList resource that specifies the font or fonts used to display it. These pairs of XmString/XmFontList resources are all you need to display arbitrary text. There are shortcuts, however: you may not always have to specify an XmFontList, and many times you may find it simpler to specify text through a regular String resource than an XmString. The following two subsections explain how you can do this. The subsections after those explain how XmStrings are printed in PostScript, and discuss resources you may need to specify for printing, and explain how to display and print internationalized text with the Plotter widget.

3.3.1 Font List Inheritance

The Plotter XmNfontList resource specifies a default font list. This resource is not directly used by itself, but its value is used as the default of all other XmFontList resources of the Plotter and all Plot subclasses. Thus, instead of specifying a separate font list for each piece of text you display, you may be able to specify all the fonts you need in a single font list resource. To do this, of course, you will need to specify appropriate font tags when you create the XmStrings to be displayed.

For example, you might specify a font list like the following:

```
*plotter.fontList: \
-*-times-medium-r-*-*-*-180-*-*-*-*-*-*=TITLE,\
-*-times-medium-r-*-*-*-140-*-*-*-*-*-*=AXIS,\
-*-times-medium-r-*-*-*-120-*-*-*-*-*-*=LABELS,\
-*-symbol-medium-r-*-*-*-140-*-*-*-*-*-*=SYMBOL\
```

Then, when you create the XmString for the plotter title, you would use the font tag "TITLE" to specify that you wanted the 18-point font to be used. In this case, there is no need to specify the XmNtitleFontList resource--the correct value is automatically inherited from XmNfontList. Similarly, when you create the XmString for the Axis titles, you'd use "AXIS" as the font tag. Or you might create a more complicated XmString that uses the "AXIS" font tag with the "SYMBOL" font tag to mix Times-Roman text with mathematical symbols and Greek letters from the Symbol font.

3.3.2 Automatically Created XmStrings

When using the Plotter widget, you may often find that the text you want to display (as the Plotter title, for example) uses only a single font. In this case, since you won't be using the multi-font capability of the XmString, having to convert the text to an XmString before displaying it is simply a nuisance. For this reason the Plotter widget and its Plot children have a String resource that corresponds to each of their XmString resources. For example, you can specify text for the Plotter's title by setting an XmString on the XmNtitleString resource, or by setting a regular NULL-terminated C String on the XmNtitle resource--it is as if the Motif XmLabel widget had a String resource named XmNlabel in addition to its XmString resource named XmNlabelString. The naming convention evident here is used throughout the Plotter library: if the resource name ends in "String", then it is an XmString resource. The resource with the same name minus the "String" specifies the same text as a regular String. (The exception to this rule are the constraint resources for specifying legend items: XmNlegendString is the XmString resource, and XmNlegendName is the String resource.)

If you only use a single font for the Plotter title, or other text, then you can specify the text as a String rather than an XmString. You still must specify the font to be used, but this is also a simple case--you can specify an XmFontList that consists of a single font. Thus you might have resource specifications like these:

```
*plotter.title: Average Weekly Temperature Extremes\n1984-1994
*plotter.titleFontList: *-helvetica-bold-o-*-*-*-180-*
```

All you need to specify is the text to display and the font to display it in.

In the previous section we noted that the default for all XmFontList resources is inherited from the Plotter XmNfontList resource, and saw how this allows a single font list specification to be used for all the text in the Plotter and in all of its Plot children. Assuming we want to use different fonts in different parts of the Plotter display, then a single font list specification of this sort will require different font list tags to identify the various fonts in the XmFontList. This implies that it is no longer sufficient to simply specify text as a String--we also need to specify a font list tag to indicate which font from the font list will be used to display the text.

Each String text resource in the Plotter and its Plot children is accompanied by a pair of other resources which combine to produce a font list tag for the text. For XmNtitle, for example, these two resources are XmNtitleStyle (a string) and XmNtitleSize (an integer). These two resources specify a font list tag as follows: if both are specified, they are concatenated to produce a tag like "bold14" or "italic12". If a style is specified, but the size is zero, then only the style is used--"italic" or "bold", for example. Similarly, if the style is NULL and the size is non-zero, then only the size is used, producing tags like "10" or "18". Finally, if no style is specified, and the size is set to zero, then the default tag, XmFONTLIST_DEFAULT_TAG, is used.

This system of font tag selection based on style and size resources is designed for use with font lists like the following:

```
-*-helvetica-medium-r-*-*-*-120-*-*-*-*-*-*=12,
-*-helvetica-medium-r-*-*-*-100-*-*-*-*-*-*=10,
-*-helvetica-medium-r-*-*-*-140-*-*-*-*-*-*=14,
-*-helvetica-medium-r-*-*-*-180-*-*-*-*-*-*=18
```

In fact, this font list is the default value of the Plotter XmNfontList resource, and thus, through font list inheritance, the default value for all Plotter and Plot subclass XmFontList resources. This default font list was chosen, of course, to work well with the default values of all the style and size resources in the Plotter library. As you can see, style resources throughout the library all have NULL as their default, and thus are unused in the font list tags. Similarly, size resources throughout the library are one of the four numbers 10, 12, 14 and 18. As you can see, the default font list will not work adequately if you want to specify any style resource, or want to specify a size resource that is not one of these four numbers.

While this system of specifying a large font list with many fonts, and then constructing font list tags from a style and a size specification may seem confusing and cumbersome, consider the advantages it brings: only a single XmFontList specification is needed, and once an appropriate font list is specified, all single-font text can be specified as a String rather than an XmString, and the font to be used for that text can be specified (and modified) through simple string and integer resources. If you choose to use this scheme (instead of specifying many single-font font lists, as described above), then you will end up with resource specifications like those shown in Example 3-1.

Example 3-1. Specifying text style an size with a single font list

```
    *plotter.fontList: \
    -*-helvetica-medium-r-*-*-*-120-*-*-*-*-*-*=12,\
    -*-helvetica-medium-r-*-*-*-100-*-*-*-*-*-*=10,\
    -*-helvetica-medium-r-*-*-*-140-*-*-*-*-*-*=14,\
    -*-helvetica-medium-r-*-*-*-180-*-*-*-*-*-*=18,\
    -*-helvetica-medium-o-*-*-*-140-*-*-*-*-*-*=italic14,\
    -*-helvetica-bold-r-*-*-*-140-*-*-*-*-*-*=bold14,\
    -*-helvetica-bold-o-*-*-*-140-*-*-*-*-*-*=bold-italic14,\
    -*-helvetica-bold-r-*-*-*-180-*-*-*-*-*-*=bold18

    *plotter.title: Average Weekly Temperature Extremes\n1984-1994
    *plotter.titleStyle: bold
    *plotter.titleSize: 18

    *plotter.legendTitle: Key
    *plotter.legendTitleStyle: italic
    *plotter.legendTitleSize: 14

    *plotter.xaxis.title: Week
    *plotter.yaxis.title: Temperature (degrees C)
    *plotter.XiAxis.titleStyle: bold
    *plotter.XiAxis.titleSize: 14

    *plotter.XiAxis.labelSize: 12
```

In summary, note first that nothing discussed in this section matters if you choose to create your own XmStrings--in that case it is up to you to use appropriate font list tags and font lists that define those tags. On the other hand, if you do not need multiple fonts in your displayed text, you may prefer to specify simpler String resources and have the Plotter library convert those Strings to XmStrings. In this case there are two different approaches you can take. The first approach is to specify the single font you want on each of a number of distinct XmFontList resources. The second approach is to specify a single large font list on the XmNfontList resource. The other XmFontList resources will inherit this font list, and you will use the various style and size resources to specify font tags, which in turn specifies which fonts in the font list to use for the various pieces of displayed text. Note, of course, that it is also possible to use a combination of these two approaches.

3.3.3 PostScript Font Lists

One of the downfalls of the X Window System is that X fonts are not compatible with PostScript fonts. There is no general way to determine a PostScript font that corresponds to a given X font. The Plotter library knows about the five most common families of X fonts, and can determine PostScript equivalents for them. This means that the Plotter can automatically print any text you display in Adobe Helvetica, Times, Courier, New Century Schoolbook, or Symbol, in plain, italic, bold, or bold-italic.

If you use any fonts outside of these standard five font families, however, you must tell the Plotter what PostScript font to print them with--just as you specify X fonts for text display with an XmFontList, you must also specify the equivalent PostScript fonts with an XiPSFontList. An XiPSFontList is an array of XiPSFontRec structures:

```
typedef struct {
    String name;
    int pixelsize;
} XiPSFontRec, *XiPSFontList;
```

The name field specifies the name of the PostScript font to use, which will be a string like "Utopia-Bold". The pixelsize field specifies the size, in pixels, at which the corresponding X font is used. (Pixel size is used instead of point size for this field because it is independent of the display resolution for which an X font is defined.)

Every XmFontList resource has a corresponding XiPSFontList resource. When you specify an XiPSFontList, it must have the same number of entries as the XmFontList it corresponds to, and the entries must occur in the same order. When you specify an XiPSFontList resource, the Plotter library does not copy the array of XiPSFontRec structures, nor does it copy the font name strings in the array, so these should be in static memory, or in allocated memory that will not be modified or freed for the lifetime of the Plotter widget.

You might specify an XiPSFontList from C code as follows:

```
static XiPSFontRec default_ps_font_list[] = {
    {Utopia-Regular, 12},
    {Utopia-Regular, 14},
    {Utopia-Italic, 14},
    {Utopia-Bold, 18}
};
```

The Plotter library defines a String-to-XiPSFontList resource converter so that you can specify XiPSFontList resources from a resource file as a comma-separated list of name/size pairs. For example, you could specify the same XiPSFontList as above like this:

```
*plotter.psFontList:\
Utopia-Regular 12, Utopia-Regular 14,\
Utopia-Italic 14, Utopia-Bold 18
```

Remember that you only need to specify an XiPSFontList if you are using fonts outside of the five "standard" X font families. Also, not that specifying a PostScript font in an XiPSFontList is not sufficient to guarantee that it will be printed--you have to be sure that your printer supports the font you want!

If you use a font that is not in one of the five font families that the Plotter library automatically knows about, and if you do not specify an XiPSFontList, or if you specify a NULL name in the XiPSFontRec structures, then the Plotter widget will print a warning message when it attempts to print. It does not fail altogether, however. In this case, when it cannot determine the appropriate PostScript font to use, it uses the X font and prints a bitmap representation of text in that font. On a high-resolution printed page, this bitmap text will be noticeably low-resolution, but it is better than having no text displayed at all. This can be useful when you simply do not have a PostScript font that corresponds to the X fonts you are using.

When you do specify a PostScript font in an XiPSFontList, you should specify a font and size that match the X font exactly, or at least match closely. The Plotter library assumes that the individual characters in the X font have the same size as the characters in the PostScript font, and therefore that text will take up the same amount of room when printed as it does on the screen. If, for some reason, the text comes out larger in PostScript than it does in X, then its width and/or height will be shrunk to fit the space allocated for it. Thus, if you display text with a narrow font like Helvetica in X, but use an XiPSFontList to tell the Plotter widget to print that text with a wide font like New Century Schoolbook, then the New Century Schoolbook text will be shrunk in width to fit, and will come out looking like a condensed version of the font was used.

One final note about X and PostScript font encodings is necessary. X fonts that display the standard Latin characters are all encoded in the ISO8859-1 international standard. When they designed the PostScript language, however, Adobe chose its own encoding (known in PostScript as "StandardEncoding") for these fonts. ISO8859-1 and PostScript StandardEncoding are the same for all standard characters used in English, but they differ for accented characters and other Latin characters used in European languages. The PostScript code output by the Plotter widget handles this situation correctly--it checks each font it uses to see if it has the StandardEncoding encoding. If so, it re-encodes the font to use ISO8859-1. Thus, any ISO8859-1 text displayed in X will also be correctly displayed in hardcopy. Note that StandardEncoding and ISO8859-1 are the only two encodings that the Plotter library has special knowledge of; in all other cases, the X font and the corresponding PostScript font must use identical encodings.

3.3.4 Internationalized Text Display

To display text in certain languages, more than 256 distinct character glyphs are required, and so a 16-bit font is used, and strings are represented as arrays of 2-byte characters. Since the Plotter widget uses XmStrings for its text display, and because Motif XmStrings support internationalization, the Plotter widget can display this kind of 16-bit text. The Plotter widget can also print 16-bit text, assuming, of course, that an appropriate PostScript font exists on your printer. In PostScript, all 16-bit fonts are implemented as "composite" or "Type 0" fonts. As noted in the previous section, the Plotter widget relies on the assumption that the X font and the PostScript font have exactly the same encoding.

Some languages use more than a single font to display text. Japanese text display, for example, may require a 16-bit Kanji font, as well as 8-bit Katakana, Hiragana and Latin fonts. There are two ways to handle text display when multiple fonts are required like this. The first is to define an XmFontList with a separate font tag for each individual font, and to use an XmString which explicitly switches from one font tag to another each time the text switches fonts. This was the only method to display this sort of text prior to Motif 1.2. To print text of this sort, you need simply to define an XiPSFontList that corresponds to the XmFontList--each X font must have a corresponding PostScript font with exactly the same encoding. In Japanese, where a 16-bit Kanji font is required, a 16-bit PostScript composite font must be specified to match.

The problem with the above approach is that it is not a particularly natural one. Text in a language like Japanese is not typically broken up into a bunch of discrete segments; instead it is encoded as a single string of text, which uses special embedded codes to indicate when the font should switch. With the advent of X11R5 and Motif 1.2, there is another technique for displaying text that uses these higher-level meta-encodings. X11R5 defines the XFontSet abstraction, which is a collection of all the fonts necessary to display text in a given language. In Motif 1.2, you can define an XmFontList that contains a single XFontSet, instead of separately specifying each of the individual fonts. When you define an XmFontList that contains an XFontSet, then your XmString can contain a single string of text with the appropriate meta-encoding of embedded codes to switch among the fonts defined in the XFontSet. When the application is properly internationalized, Xlib will understand the meta-encoding that is in use, and will be able to display the text using appropriate fonts from the XFontSet.

Since the Plotter widget is based on Motif 1.2, it can, of course, display text represented in this way. Printing that text may be more complicated, however. Once again, the single basic requirement is that you provide (in an XiPSFontList) a PostScript font with an identical encoding to the X font. In this case, you must specify a composite PostScript font with a meta-encoding that matches the meta-encoding used by the XFontSet. The encodings of the individual fonts within the composite PostScript font must also match the encodings of the X fonts within the XFontSet, of course. For some languages and some meta-encodings, such an composite PostScript font may exist. For other languages or meta-encodings, there may be no appropriate PostScript font. In this case, you will have to break your text up into separate segments, one for each font, as described above, and as you would have done in a Motif 1.1 application.

3.4 Markers

The LinePlot widget, and its subclass, the ErrorPlot can optionally mark each data point in the plot with a small glyph. The XmNmarker resource of these widgets specifies the glyph to use, and is of type XiMarker. An XiMarker is an abstraction that can display a glyph in X and in PostScript. The Plotter library contains a number of predefined markers, and Chapter 17, *Defining Custom Markers* explains how you can define custom markers of your own for specialized graphs.

The Plotter library defines 12 different marker glyphs each in four different sizes, for a total of 48 distinct markers. These predefined markers are pictured in Figure 3-1and listed in Table 3-1. (You can also find these pre-defined markers listed in the quick reference section at the end of this manual.)

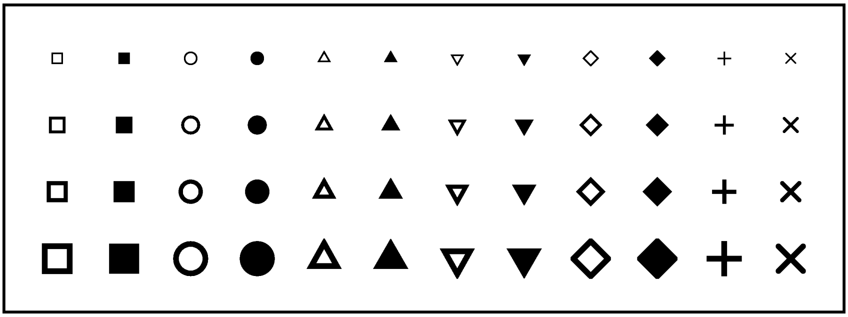


Figure 3-1. Predefined markers

The markers shown in the table are declared in the header file <*Xi/Plotter.h*>. You can use the symbols exactly as shown in your C code to set the XmNmarker resource. The names of these markers are fairly self-explanat-ory. The "Triangle" markers are equilateral triangles with one vertex pointing up, and the "Triangle2" markers are the same triangle rotated 180 degrees, with one vertex pointing down.

Table 3-1. Predefined Markers

Tiny Markers	Small Markers
XiMarkerTinySquare	XiMarkerSmallSquare
XiMarkerTinyFilledSquare	XiMarkerSmallFilledSquare
XiMarkerTinyCircle	XiMarkerSmallCircle
XiMarkerTinyFilledCircle	XiMarkerSmallFilledCircle
XiMarkerTinyTriangle	XiMarkerSmallTriangle
XiMarkerTinyFilledTriangle	XiMarkerSmallFilledTriangle
XiMarkerTinyTriangle2	XiMarkerSmallTriangle2
XiMarkerTinyFilledTriangle2	XiMarkerSmallFilledTriangle2
XiMarkerTinyDiamond	XiMarkerSmallDiamond
XiMarkerTinyFilledDiamond	XiMarkerSmallFilledDiamond
XiMarkerTinyPlus	XiMarkerSmallPlus
XiMarkerTinyX	XiMarkerSmallX
Normal Markers	**Large Markers**
XiMarkerSquare	XiMarkerLargeSquare
XiMarkerFilledSquare	XiMarkerLargeFilledSquare
XiMarkerCircle	XiMarkerLargeCircle
XiMarkerFilledCircle	XiMarkerLargeFilledCircle
XiMarkerTriangle	XiMarkerLargeTriangle
XiMarkerFilledTriangle	XiMarkerLargeFilledTriangle
XiMarkerTriangle2	XiMarkerLargeTriangle2
XiMarkerFilledTriangle2	XiMarkerLargeFilledTriangle2
XiMarkerDiamond	XiMarkerLargeDiamond
XiMarkerFilledDiamond	XiMarkerLargeFilledDiamond
XiMarkerPlus	XiMarkerLargePlus
XiMarkerX	XiMarkerLargeX

When you create a LinePlot widget, a resource converter is automatically registered for the XiMarker type. This means that you can specify the XmNmarker resource for any LinePlot or ErrorPlot widget from a resource file. The converter recognizes the names shown in Table 3-1 with the "XiMarker" prefix removed. For example, you could set the marker for a line plot with a resource specification like this:

```
lineplot1.marker: SmallFilledTriangle
```

The converter is case-sensitive, so you must specify the names with mixed-case, as they are shown.

3.5 Fill Patterns

The BarPlot and PiePlot widgets can fill their bars and wedges with a solid color, or with a fill pattern. Both of these widgets have resources of type XiFillPattern. Like the XiMarker described above, the XiFillPattern is an abstraction that can display a fill pattern both in X and in PostScript. The file <*Xi/Plotter.h*> declares 25 pre-defined fill patterns that you can use directly in your C code. Table 3-2 lists these pre-defined patterns, and Figure 3-2 displays them. (For later reference, you can also find the table of predefined patterns in the quick reference section at the end of this manual.)

Table 3-2. Predefined fill Patterns

LeftHatch	**RightHatch**	**Crosshatch**
XiFillPatternLefthatch1	XiFillPatternRighthatch1	XiFillPatternCrosshatch1
XiFillPatternLefthatch2	XiFillPatternRighthatch2	XiFillPatternCrosshatch2
XiFillPatternLefthatch3	XiFillPatternRighthatch3	XiFillPatternCrosshatch3
XiFillPatternLefthatch4	XiFillPatternRighthatch4	XiFillPatternCrosshatch4
XiFillPatternLefthatch5	XiFillPatternRighthatch5	XiFillPatternCrosshatch5
Grid	**Gray**	**Special**
XiFillPatternGrid1	XiFillPatternGray1	XiFillPatternNone
XiFillPatternGrid2	XiFillPatternGray2	XiFillPatternSolid
XiFillPatternGrid3	XiFillPatternGray3	
XiFillPatternGrid4	XiFillPatternGray4	
XiFillPatternGrid5	XiFillPatternGray5	

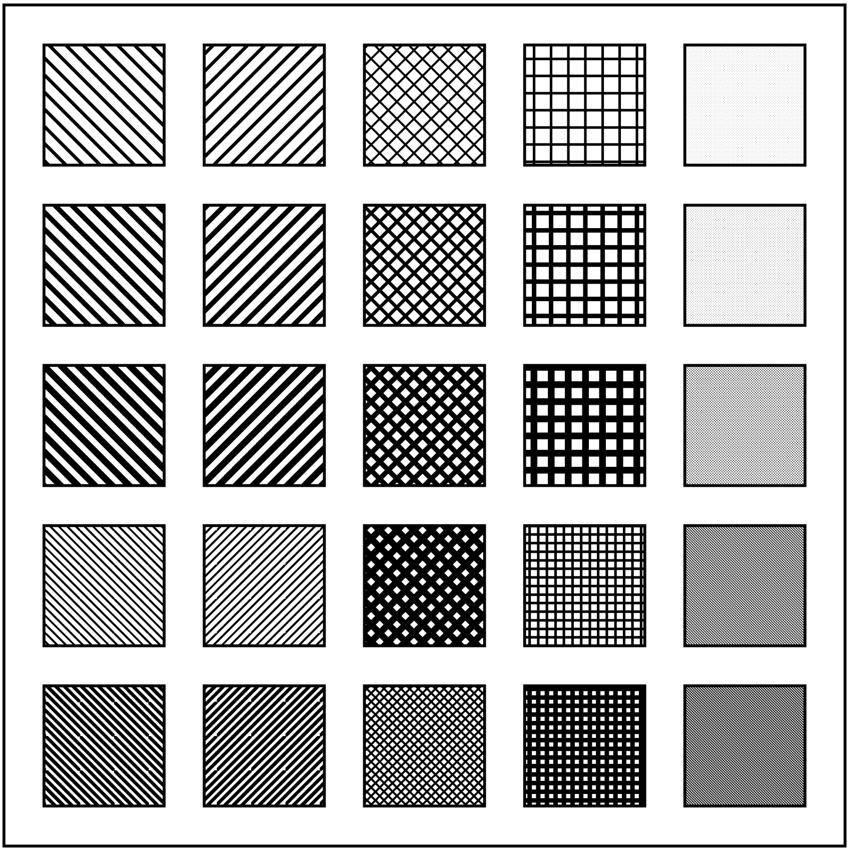


Figure 3-2. Predefined fill Patterns.

If none of these predefined patterns is appropriate for your application, you can define custom fill patterns of your own.Chapter 18, *Defining Custom Fill Patterns* explains how to do this.

You can use fill patterns in your C code exactly as listed in Table 3-2.

```
    XtVaSetValues(barplot, XmNfillPattern, XiFillPatternGray2, NULL);
```

You can also specify fill patterns from a resource file. When you create a BarPlot or a PiePlot, this widget automatically registers a resource converter for the XiFillPattern type. This resource converter recognizes the names listed in Table 3-2 with the "XiFillPattern" prefix removed, and all uppercase letters converted to lowercase.[1]

There are three special predefined fill patterns that do not fall into one of the five basic categories listed above. They are the following:

XiFillPatternNone

> This value specifies that the bar of a BarPlot or the wedge of a PiePlot should not be filled at all, and that whatever is "underneath" the plot (grid lines, for example) should show through.

XiFillPatternSolid

> This value specifies that an area should be filled with a solid color rather than with any kind of pattern. (The color to use is specified with some other resource.) This is often the default value of any XiFillPattern resource.

3.6 Line Patterns

The LinePlot widget allows you to specify a dash pattern to use when drawing a line that connects the data points, and the Axis widget allows you to specify line patterns for the lines of the grid it draws in the background of the Plotter. These widget resources have a representation type of XmRLinePattern, but there is no corresponding XiLinePattern type; line patterns are represented with a character array--a String.

The file <*Xi/Plotter.h*> declares some convenient pre-defined line patterns, which are shown, along with two custom line patterns in the LinePlots of Figure 3-3.

Line patterns are specified as a list of numbers that specify the number of pixels to draw and the number of pixels to skip. Since these numbers are always small integers, they fit conveniently into the char type and arrays of numbers are easily specified with string constants. In C code, you can use the "\nnn" escape to encode line patterns into strings. Thus, a dash pattern of ten pixels on followed by six pixels off would be represented by the string "\012\006". (Remember that the "\nnn" escape uses octal numbers--012 in base 8 is 8+2, or 10, in base 10). Line patterns will generally have an even number of elements, but they can be any length, even or odd. When each element in the pattern has been drawn, the pattern starts over at the first element. A dot-dash pattern, for example will generally have four elements: "\001\003\004\003"--one pixel on (a dot), three off, four on (a dash), and another three off.

[1] For historical reasons, the XiMarker converter and the XiFillPattern resource converters use different capitalization conventions of the values they convert. This can unfortunately be a source of confusion.

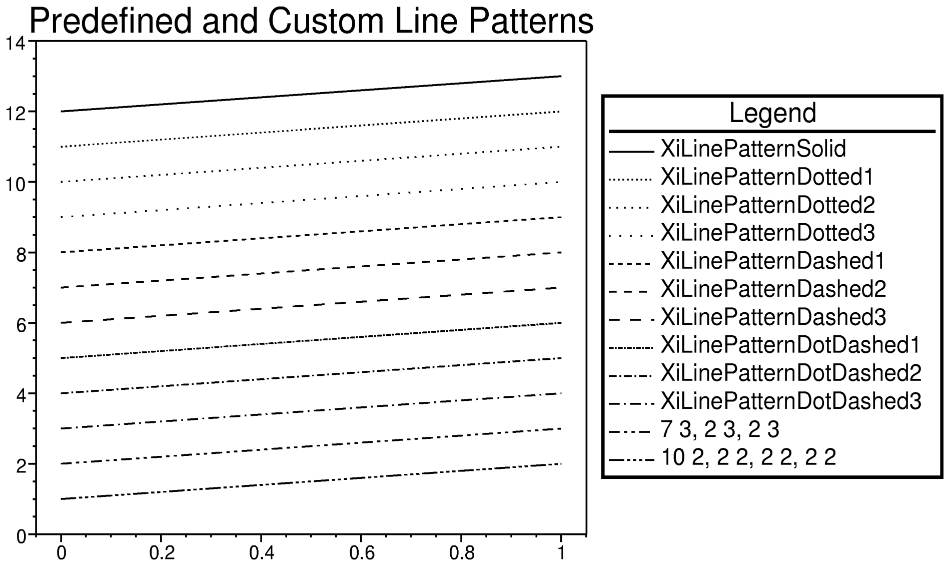


Figure 3-3. Predefined Line Patterns

Figure 3-2 shows the definitions of each of the predefined line patterns shown in the previous figure. These definitions are internal to the Plotter library and are not visible to applications. The pattern names are declared, however, in the header *<Xi/Plotter.h>*. A table of these pattern names also appears in the quick reference section.

```
    char *XiLinePatternSolid = NULL;
    char *XiLinePatternDotted1 = "\001\002";
    char *XiLinePatternDotted2 = "\001\004";
    char *XiLinePatternDotted3 = "\001\006";
    char *XiLinePatternDashed1 = "\003";
    char *XiLinePatternDashed2 = "\005";
    char *XiLinePatternDashed3 = "\007";
    char *XiLinePatternDotDashed1 = "\003\001\001\001";
    char *XiLinePatternDotDashed2 = "\005\002\001\002";
    char *XiLinePatternDotDashed3 = "\007\003\001\003";
```

Figure 3-4. Definitions of the standard line patterns

You can use any of these predefined patterns in your C code, patterns with a string constant. In either case, the widget will make a copy of the pattern you specify, as it does with all String resources. The cautions described above for String resources apply here. Note that XiLinePatternSolid is simply defined to be NULL. You can use NULL for any line pattern resource to get a solid line.

The Plotter library automatically registers a resource converter for line pattern resources. This converter recognizes the names of the line patterns shown in Figure 2-4 with the "XiLinePattern" prefix removed. It will also convert arbitrary line patterns specified as a list of integers (in base 10, not base 8) separated by spaces. So you could use resources like the following to specify one pre-defined and one custom line pattern for the major and minor grid lines of an Axis widget:

```
*plotter.xaxis.gridPattern: Dotted1
*plotter.xaxis.subgridPattern: 2 5
```

3.7 Line Widths

Throughout the Plotter library there are widget resources that specify the width of lines. The values of these resources are used directly to set the line_width field in an Xlib Graphics Context (GC). Line widths in Xlib are always measured in pixels, but zero is a special value. If you specify a line width of zero, Xlib will draw lines one pixel wide, using a fast line drawing algorithm. Zero-width lines are a special case that allow the server to relax some of the constraints placed on its drawing of wide lines. For example, if you draw a zero-width line from point A to point B, the X server might set a slightly different set of pixels than it would set if you had drawn the line from point B to point A. With wider lines, this can be unacceptable, and the X server is not allowed to do it, but with lines that are only one pixel wide, the difference is rarely noticeable.

In the Plotter library, when you want a narrow one pixel wide line, it is generally fine to specify a line width of 0, and this is in fact the default for a number of Plotter, Axis and Plot widget resources. In most cases, a line width of 0 will not be noticeably different than a line width of 1, except that it will be drawn faster.

Although it is important to understand the difference between line widths of zero and one, note that lines one pixel wide are quite narrow, and, unless you are displaying plots in a very small Plotter widget, a line width of 2 or 3 pixels for line plots, axes, and so on may give you better looking results.

3.8 Colors

Many of the widget resources in the Plotter library specify colors for text, lines, patterns, and other things that appear within the Plotter window. The Plotter widget does not handle color resources specially--these are standard color resources, of type Pixel. A Pixel is not a simple type that can be expressed with a constant value in C, as a double or a String can be; Pixel values must be allocated with XAllocNamedColor() or some similar function, and must be freed when no longer necessary. Because this can be somewhat cumbersome, and because the Plotter widget has so many color resources, this section describes some easier ways of handling colors in an Xt application.

The easiest way to specify a color is to do so in a resource. file. This way, the Xt resource converters lookup the color name and allocate the Pixel for you. The Xt resource converter mechanism also caches the converted value for efficient reuse, and frees it when all the widgets that use it are destroyed; this takes a large housekeeping burden off the application. Another advantage to using a resource file is that the colors are then customizable to the end-user's preferences (or for monochrome hardware).

There are some applications, however, for which you do not want to rely on a resource file to be correctly installed, or for which you do not want to allow user customizability of the colors. In this case you need to hardcode the color values. Instead of allocating and freeing the Pixel values yourself, though, you can still rely on the Xt resource converters to do this for you using XtVaSetValues(). This technique uses the special XtVaTypedArg argument that was also used above for setting double values:

```
XtVaSetValues(lineplot,
XtVaTypedArg, XmNlineColor, XmRString, "maroon", 7,
XtVaTypedArg, XmNmarkerColor, XmRString, "navy", 5,
XtVaTypedArg, XmNshadowColor, XmRString, "gray", 5,
NULL);
```

The four arguments following XtVaTypedArg specify the name of the resource to be set, the type that the resource is to be converted from, the value that is to be converted, and the size of the value to be converted. When converting from strings (which is what you will almost always do), the length argument should be the strlen() of the string, plus one, for the terminating '\0' at the end of the string. Note that you can use this XtVaTypedArg technique with XtVaCreateWidget() as well as with XtVaSetValues().

3.9 Cursors

The Plotter widget has a number of resources of type Cursor. Like Pixel values, Cursor values are X server objects that need to be allocated and freed. The default values for these resources are all quite reasonable, so you may never need to set them yourself. If you do, however, you can either allocate and free the Cursor values yourself by calling Xlib functions, or, as with Pixel values above, you can have the Xt resource converter mechanisms do this for you.

As with Pixel values, the best ways to set cursor resources are in resource files, or with an XtVaTypedArg argument to XtVaSetValues or XtVaCreateWidget(). The String-to-Cursor converter recognizes the names of the cursors in the standard cursor font. You can view this font with:

```
    xfd -fn cursor
```

The names of the cursors appear in the file <*X11/cursorfont.h*> which defines symbols for each of the glyphs in the font. The resource converter recognizes the names shown in this file, with the "XC_" prefix stripped off.

The Plotter widget registers its own enhanced String-to-Cursor converter which recognizes all of the standard cursor names, and also allows you to specify a foreground and background color for your cursors. To use it, simply follow the name of the desired cursor with the optional foreground and background color names. For example, this line in a resource file:

```
    *plotter.plotterCursor: crosshair red gray
```

specifies the "crosshair" cursor with a red foreground on a gray background. If you do not specify the foreground and background colors, the converter will use a black foreground and a background inherited from the XmNbackground resource. (A note about cursor "backgrounds": most cursors are mostly transparent, so you can see what is under them. The background color is usually only used in a thin strip around edge of the cursor shape, so that the cursor is visible even when it is over a region the same color as the foreground.)

While there are a lot of useful cursors in the standard font, many of them are frivolous ("XC_gumby" and "XC_trek" for example), and some are downright stupid ("XC_bogosity"). Serious applications may want to use cursors that do not appear in this font. You can do this with the Xlib functions XCreateGlyphCursor() which makes a cursor from two glyphs of a specified font, XCreatePixmapCursor() which creates a cursor from two bitmaps. (One of the glyphs or bitmaps is a mask that defines the shape of the cursor--this is necessary because cursors are non-rectangular.)

3.10 Inherited Resource Values

Almost all color resources in the Plotter, Axis, and all the Plot widgets inherit their values from one of the three Plotter resources XmNforeground, XmNbackground, or XmNplotAreaColor. The XmFontList and XiPSFontList resources of the Axis, PiePlot, and TextPlot widgets are similarly inherited from the XmNfontList and XmNpsFontList resources of the Plotter. An inherited resource is one that gets its default value from some other resource. If you do not set the XmNmarkerColor resource of a LinePlot widget, for example, that resource will be set to whatever value is set on the XmNforeground resource of the Plotter. If the XmNforeground resource was itself unset, then XmNmarkerColor will inherit the default value of XmNforeground. This scheme makes it easy to specify the color of all the various parts of the Plotter widget without setting every color resource.

There are a couple of things to be aware of about these inherited resources, however. First, note that this inheritance works only when a widget is created, not when resource values are set. In the example above, the LinePlot will inherit the value of XmNforeground that is in effect when it is created, but future changes to the Plotter XmNforeground resource will not be propagated to it. So although you can use resource inheritance to set initial colors across the Plotter widget, you cannot use it to change colors across the widget--to do that you must set all the color resources individually.

4

Plot Data Representation

Before any kind of Plot widget can plot data in its parent Plotter widget, you must specify the data values it is to display. Every application will likely have its own particular data structures for plot data--some will use double values, and others may use float or unsigned short values, for example. Some applications will find it most convenient to store (x,y) pairs for a line plot in two separate arrays of points, and others will prefer to store these values in a single array of structures. Applications that query complex data from a database, for example, may want to plot data values stored in the middle of much larger records of data.

All of the Plot widget types that require an array of values to plot are flexible enough to handle each of the cases described above. Each plot type provides convenience functions for easily setting data that is stored in some of the most common representations (arrays of double or int, for example), but also provide resources that allow a very flexible method of specifying data. For each required array of values, the Plot widget is given a pointer to an array of arbitrary structures, each of which contains a data value for the Plot. A resource specifies the size of each element of this array so that the Plot widget can step from one element of the array to the next. The Plot widgets can accept data of type double, float, int, and a number of other signed and unsigned scalar types. Other resources specify the type of the data value and the offset from the beginning of the data structure at which it is stored. With this information, (size of the structure, and the type and offset of the value within the structure) the widget can extract the data values it is interested in from whatever structures are most convenient for the application; there is no need to make special copies of data to be plotted.

Note that although values can be extracted from arbitrary structures with this data representation scheme it is also of course possible to specify data as a simple array of values. This is just a degenerate case: when each element of the array is a simple data value, then the size of this "structure" is the size of the value, and the offset of the value within this structure is always zero.

Also note that the structures in the array may well contain more than one data value--you can pass the same array pointer to specify the X data points and the Y data points for a line plot, for example, and simply specify different offsets (and perhaps different types as well) for the X and Y values within each structure.

The variables that specify structure size, data type, offset, and so on are all contained in the XiPlotData structure. The following section describes this XiPlotData abstraction in more detail, and the sections after that give examples of some common ways you might choose to specify plot data.

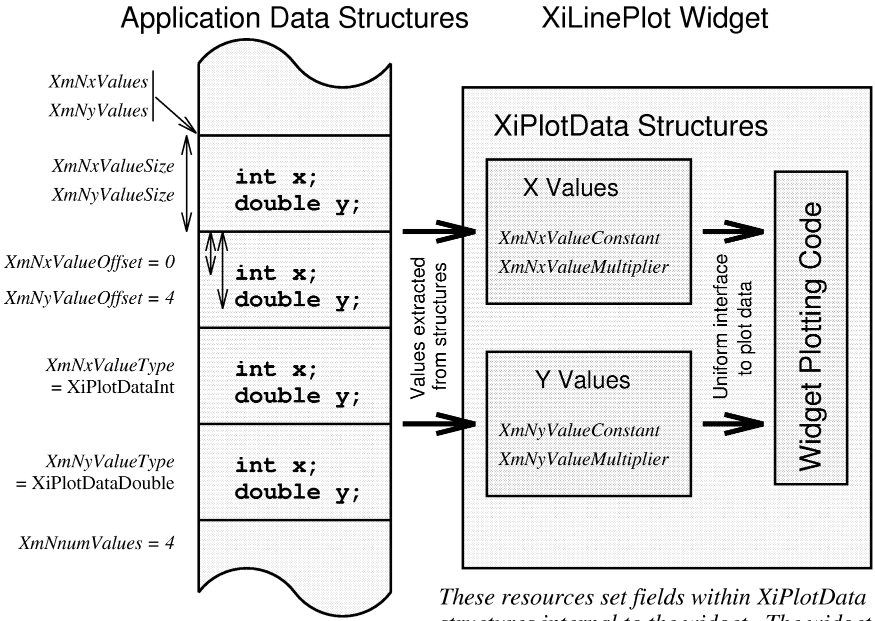
4.1 The XiPlotData Abstraction

An XiPlotData abstraction is a structure that contains all the information necessary to extract values from an array of arbitrary structures. Almost all Plot widget classes have one or more of these structures: the BarPlot widget, for example, has one XiPlotData structure that specifies how to obtain the Y values of each bar. The LinePlot widget has XiPlotData structures that specify how to obtain both the X and Y values of its points and the ErrorPlot widget adds two more XiPlotData structures to specify the high and low bounds of the error bars.

Figure 4-1 shows the definition of the XiPlotData structure, It is quite important to understand how this structure works, but note that you will never have to create or manipulate one yourself--the Plot widgets all provide resources and functions with which you can set all the fields.

```
typedef enum {
typedef struct {    XiPlotDataDouble,
    XtPointer values;    XiPlotDataFloat,
    Cardinal num;    XiPlotDataUnsignedLong,
    Cardinal size;    XiPlotDataLong,
    Cardinal offset;    XiPlotDataUnsignedInt,
    unsigned char type;    XiPlotDataInt,
    double constant;     XiPlotDataUnsignedShort,
    double multiplier;    XiPlotDataShort,
} XiPlotData;    XiPlotDataUnsignedChar
} XiPlotDataType;
```

Figure 4-1. The XiPlotData structure and the XiPlotDataType enumerated type



The application stores its data in arrays external to the widget, using whatever data structures are most convenient for it. It sets resources on the widget to point to the arrays, specify the data type, and describe how the data is stored in the structures. Other resources perform an optional linear transform on the data.

These resources set fields within XiPlotData structures internal to the widget. The widget plotting code uses the XiPlotData abstraction to extract plot data values from the application structures in a uniform way, regardless of data type or structure size.

Figure 4-2. Specifying plot data through XiPlotData resources

The paragraphs below explain each of the fields of the XiPlotData structure, and Figure 4-2 diagrams the meaning of the fields and illustrates how they allow Plot widgets to extract data from application structures. The figure uses the LinePlot widget as an example; the LinePlot resource names are all obviously based on the XiPlotData field names.

values

This field is the pointer to the array of structures (or array of values) from which the plot data values are to be extracted. It is set through resources like the BarPlot

XmNvalues and the LinePlot XmNxValues and XmNyValues. Note that it is an untyped pointer (an XtPointer).

num

This field specifies the number of elements in the array specified by the values field. You usually set it through a resource named XmNnumValues. Note that when a Plot widget, like the LinePlot takes more than one set of data values (X values and Y values, in this case), there is generally only one resource that sets this num field for all XiPlotData structures. This is because plots like the LinePlot always require that there be the same number of values in each of their arrays. Be careful when setting this field that you never make it larger than the number of elements actually available in the specified array. If you do, your Plot widget might attempt to read memory beyond the bounds of the array and crash with a segmentation violation.

size

This field specifies the size, in bytes, of each element of the array specified by the values field. You should use the sizeof() operator to compute this size. Note that you are specifying the size of the entire structure, not just the size of the data value within the structure. You usually specify this field through resources like the BarPlot XmNvalueSize or the LinePlot XmNxValueSize and XmNyValueSize. The default value for these fields is 8 bytes, which is the size of a double, so if your array of structures is simply an array of double, then you will not need to set these resources.

offset

This field specifies the offset, in bytes, from the beginning of each structure in the array to the data value within that structure. Use the standard Xt XtOffsetOf() macro to portably calculate this offset for any field of a structure. You usually specify this field through resources like the BarPlot XmNvalueOffset or the LinePlot XmNxValueOffset and XmNyValueOffset. The default is 0 bytes, which means that if you have specified a simple array of values that are not contained in larger structures, you need not set these resources.

type

This field specifies the type of the data value to be extracted from each element of the array specified by the values field. The legal values are the members of the XiPlotDataType enumerated type, which is shown in Figure 4-1. The meanings of each of these enumerated values is self-evident. Note that there is no XiPlotDataChar type--the C language does not specify whether the char type is signed or unsigned, so this type would be less portable than the others. To avoid this problem, XiPlotDataUnsignedChar is the only 1-byte type that exists. You usually specify the type field through resources like the BarPlot XmNvalueType or the LinePlot XmNxValueType and XmNyValueType. The default values for these resources is XiPlotDataDouble, since double is probably the most commonly used type for plot data.

multiplier

This field is a double value by which each extracted data value is multiplied. You specify this field through resources like the BarPlot XmNvalueMultiplier and the LinePlot XmNxValueMultiplier and XmNyValueMultiplier. The default value of these resources is 1.0, so this multiplication does not change the data values at all. The value -1.0 can be used to change the sign of all the plot data points, and other values can be used to scale the values. This field, and the constant field, described below, perform a linear transform on the data points: for each value x extracted from the array specified by the values field, the widget will obtain a transformed value x'

$$x' = mx + c$$

where m is the value of the multiplier field, and c is the value of the constant field.

The multiplier and constant fields are also useful when all the data points are linearly spaced (as with the X data points of many LinePlot widgets, for example). In this case, there is no need to specify an array of data values at all. When the values field is NULL, values will be computed by performing the specified linear transform on the index of the data value. Thus, when the widget requests the *ith* data value, it will get the value x:

$$x = mi + c$$

constant

This field is the double value that is the constant c in the linear transformation equations shown above--it is added to each data value after the value has been multiplied by the multiplier field. You specify this field through resources like the BarPlot XmNvalueConstant and the LinePlot XmNxValueConstant and XmNyValueConstant. The default value for these resources is 0.0, so that ordinarily this field has no effect on the values that are plotted.

4.2 Data in Arrays of Values

The most straightforward way to store data values to be plotted, when there is no particular reason for the application to store them in any other way, is in simple arrays (not in arrays of structures). In this case, you end up with code like that shown in Example 4-1.

Example 4-1. Specifying data in simple arrays

```
unsigned int data[] = { 2, 3, 5, 7, 11, 13, 17, 19 };
XtVaSetValues(barplot,
              XmNvalues, data,
              XmNnumValues, XtNumber(data),
              XmNvalueType, XiPlotDataUnsignedInt,
              XmNvalueSize, sizeof(unsigned int),
              XmNvalueOffset, 0,
              NULL);
```

Note that it not necessary to set the XmNvalueOffset resource in this case, because its default value is 0. Also, notice that if we had used an array of double, rather than an array of unsigned, in Example 4-1 we would not have had to specify the XmNvalueType or XmNvalueSize resources either--the defaults for these values were chosen to match arrays of double.

Because specifying data in simple arrays is fairly common, each Plot widget provides convenience functions that handle simple arrays of double, float, and int. If we used an array of float, instead of an array of unsigned in Example 4-1, we might have replaced the call to XtVaSetValues() with a convenience function like this:

```
float data[] = { 2.0, 3.0, 5.0, 7.0, 11.0, 13.0, 17.0, 19.0 };
XiBarPlotSetFloatValues(barplot, 0.0, 1.0, data, XtNumber(data));
```

The second and third arguments to this convenience function specify the X value for the first bar of the bar plot, and the interval between successive bars. These convenience functions provided by each widget are documented in the chapters on those widgets. The convenience functions are usually the simplest thing to use when the application has stored its data in simple arrays of common data types. There are times, however, when no convenience function will do, and you will have to (or will prefer to) set the XiPlotData resources directly. The following sections provide examples of some common situations that you might encounter.

4.3 Data in Arrays of Structures

BarPlots, as used in the examples of the previous section, have only one data value per bar--the bars are evenly spaced, and so the X values are specified with a minimum value and an increment. For other plot types, such as line plots or high-low-close plots of financial market data, each point has more than one value--an x coordinate and a y coordinate, or a high value, a low value, and a close value. Often it is most convenient to group these values together into structures that describe the complete data point. In this case, you can end up with code like that shown in Example 4-2.

Example 4-2. Specifying data in arrays of structures

```
typedef struct {
    float high;
    float low;
    float close;
} MarketData;

extern MarketData market_history[365]; /* initialized elsewhere */

XtVaSetValues(highlowplot,
XmNnumValues, XtNumber(market_history),

XmNhighValues, market_history,
XmNhighValueType, XiPlotDataFloat,
XmNhighValueSize, sizeof(MarketData),
XmNhighValueOffset, XtOffsetOf(MarketData, high),

XmNlowValues, market_history,
XmNlowValueType, XiPlotDataFloat,
XmNlowValueSize, sizeof(MarketData),
XmNlowValueOffset, XtOffsetOf(MarketData, low),

XmNcloseValues, market_history,
XmNcloseValueType, XiPlotDataFloat,
XmNcloseValueSize, sizeof(MarketData),
XmNcloseValueOffset, XtOffsetOf(MarketData, close),
NULL);
```

The most important thing to notice about this example is the use of the standard Xt macro XtOffsetOf() to compute the value of the offset field of the high, low, and close XiPlotData structures used by the widget. Also note that XmNhighValueSize, XmNlowValueSize, and XmNcloseValueSize are all specified as sizeof (MarketData), and not sizeof(double)--these resources specify the size of the structure that contains the data value, not of the individual data value itself. (The size of the data value is implicit in the data type specification--XiPlotDataFloat in this example.)

It may seem redundant in this example to have to specify each of the values market_history, XiPlotDataFloat, and sizeof(MarketData) on three separate resources. The reason for this redundancy is of course the flexibility that is possible with the

XiPlotData scheme--having all these resources allows you to specify high, low, and close values, even when they are in separate arrays, of differing data types, and embedded in structures of varying sizes.

4.4 Data of Mixed Types

You will not always use the same data type for each coordinate of a data point. If you are using the LinePlot widget to plot a measured value against time, for example, you might store the Y coordinate as a double, and the X coordinate as an integer number of seconds. In this case, you might end up with code like that shown in Example 4-3.

Example 4-3. Specifying plot data of mixed types

```
typedef struct {
    unsigned short time;
    double value;
} DataPoint;

DataPoint system_load[1000];
int num_data_points;

/* go get the plot data from somewhere */
get_load_data(system_load, &num_data_points, 1000);

XtVaSetValues(lineplot,
              XmNnumValues, num_data_points,

              XmNxValues, system_load,
              XmNxValueType, XiPlotDataUnsignedShort,
              XmNxValueSize, sizeof(DataPoint),
              XmNxValueOffset, XtOffsetOf(DataPoint, time),

              XmNyValues, system_load,
              XmNyValueSize, sizeof(DataPoint),
              XmNyValueOffset, XtOffsetOf(DataPoint, value),
              NULL);
```

Note again that we use XtOffsetOf() in this example to compute the offset of fields within their structures. In particular, we do not assume that the time field has an offset of 0, just because it is the first structure in the field. Again, XmNxValueSize and XmNyValueSize are the size of the whole structure, not of the data values within it--the size of the data values is implicit in their types. Note that we did not specify the XmNyValueType resource; its default value, XiPlotDataDouble, is already correct in this case.

4.5 Linear Data

In the previous section we used the example of plotting measured data against time. In cases like these, the measurements are often taken at fixed time intervals, and there is really no need to specify all the X data values--the first value and the interval between values is sufficient. As described above, if we specify NULL for the values field of an XiPlotData structure, then the XiPlotData abstraction will generate a linear sequence of values based on the multiplier and constant fields. So if the Y data points were all measured at a fixed interval of 10 seconds, starting at some arbitrary "time zero", we might use code like that shown in Example 4-4.

Example 4-4. Specifying linear data

```
static double yvalues[100]; /* initialized elsewhere */
static int num_values;       /* initialized elsewhere */
static double t0 = 0.0;
static double deltat = 10.0;

XtVaSetValues(lineplot,
              XmNnumValues, num_values,
              XmNxValueConstant, &t0,
              XmNxValueMultiplier, &deltat,
              XmNyValues, yvalues,
              NULL);
```

Note that XmNxValueConstant and XmNxValueMultiplier are both resources of type double, and need to be set using one of the special techniques described in Chapter 3, *Plotter Data Types and Resources*. The default value for XmNxValueConstant is 0.0, so we did not actually have to set it in this example. If our sampling interval had been 1 second, then we would not have had to specify XmNxValueMultiplier either--its default value is 1.0. For some applications, such as signal processing, the actual size of the sample interval on the X axis is often not relevant, and the X axis may often even be displayed without any kind of labels. In this case, the XiPlotData defaults are such that you can display a LinePlot without ever setting any resources for its X data points.

Notice that since the Y data values are stored in a simple array of double, we can use the default values for XmNyValueType, XmNyValueSize, and XmNyValueOffset, and need not set these resources. Because linear X points and arrays of double Y points is a fairly common arrangement, the LinePlot widget provides a convenience function for this case.

4.6 Linear Transformations of Data

We can take the LinePlot example from the previous two sections one step further, and demonstrate another use of the multiplier and constant fields of the XiPlotData structure: suppose the Y value we are plotting is a voltage between -5 and +5 volts. This voltage is periodically sampled through an analog-to-digital converter and its values are stored as 10-bit quantities between 0 and 1023. The D-to-A converter is calibrated so that it measures voltages in increments of a hundredth of a volt, and so that an input of zero volts produces an output of 512. If the voltage is sampled every 10 milliseconds, and the collected values are stored as raw 2-byte values, Example 4-5 shows how you could produce a line plot of the data with both time and voltage values scaled correctly.

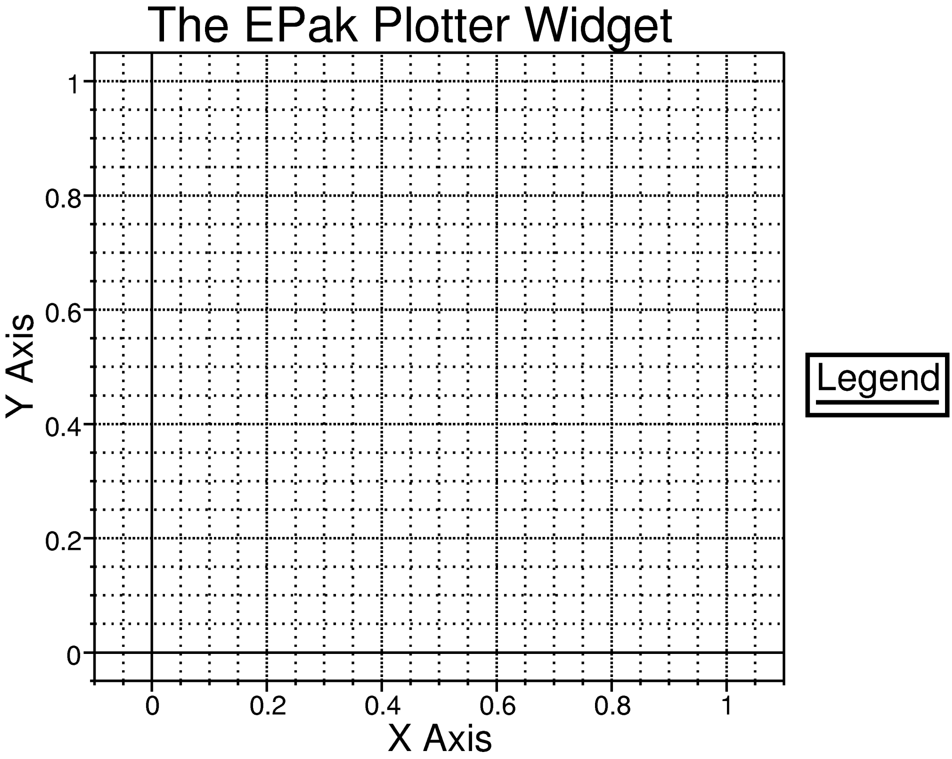
Example 4-5. Performing a linear transform on data values

```
unsigned short *data;  /* a large array of values; we'll only plot
some. */
int first_point = 200; /* the first value we plot */
double start_time = first_point * 10.0;

XtVaSetValues(lineplot,
              XmNnumValues, 100,
              XmNxValueConstant, &start_time
             XtVaTypedArg, XmNxValueMultiplier, XmRString,"10.0", 5,

              XmNyValues, &data[first_point],
              XmNyValueType, XiPlotDataUnsignedShort,
              XmNyValueSize, sizeof(unsigned short),
             XtVaTypedArg, XmNyValueMultiplier, XmRString, ".01", 4,
             XtVaTypedArg, XmNyValueConstant, XmRString, "-5.12", 6,
              NULL);
```

Notice that this example shows two different ways to set resources of type double. Note also that there is no array specified for the X values; they are always at fixed intervals. We had to specify the XmNyValueType and XmNyValuesSize resources because we did not use an array of double, but we did not have to specify XmNyValueOffset because the data is in a simple array rather than an array of structures.

4.7 String Data

With some plot types, notably BarPlots, you often want the X axis to appear with strings as the axis labels rather than numbers. To do this, you must specify an array of strings for the Axis. The Axis widget uses a scheme similar to the XiPlotData structure to allow you to specify strings from within an array of structures: the Axis XmNlabelRecordSize and XmNlabelRecordOffset resources specify the structure size and the position of the string within the structure; they serve analogous purposes to the BarPlot XmNvalueSize and XmNvalueOffset resources. These Axis resources are documented in Chapter 6, *The Axis Widget*. In most cases, you will probably use a simple array of strings when labeling an Axis, and will use an Axis convenience function to specify them. If necessary, however, you can use these resources to specify how the Axis widget should extract the strings from an arbitrary larger structure.


The EPak Plotter Widget
Y Axis
X Axis
Legend
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

5

The Plotter Widget

With the preliminaries of the previous chapters out of the way, we can finally describe the XiPlotter widget itself. The figure on the facing page shows a Plotter widget with title, an empty legend, X and Y axes, and an empty plotting area. This chapter describes the resources that control these various areas and features of the Plotter, and describes the public functions that operate on the Plotter.

All of the plots that would appear in the Plotter widget are implemented as separate widget classes, which are documented in the following chapters. The axes are also implemented as separate widgets which are documented in Chapter 6, *The Axis Widget*.

Plotter Synopsis

Class Name: XiPlotter

Class Hierarchy: Core → Composite → Constraint → XmManager → XmBulletinBoard → XiPlotter

Header File: *<Xi/Plotter.h>*

Class Pointer: `xiPlotterWidgetClass`

Constructor:

```
XiCreatePlotter(Widget parent, String name,
                ArgList args, Cardinal
                num_args);
```

5.1 Creating an XiPlotter Widget

You can create a Plotter widget with the function XiCreatePlotter(), which is a standard Motif-style widget constructor. If you prefer to use XtCreateWidget() or XtVaCreateWidget(), the class pointer for the Plotter widget is XiPlotterWidgetClass. Both the constructor function and the widget class pointer are declared in the header file <*Xi/Plotter.h*>.

Example 5-1 demonstrates both methods of creating the widget. Don't forget that you must manage any widgets you create with XtManageChild() or XtManageChildren().

Example 5-1. Creating XiPlotter widgets

```
#include <Xi/Plotter.h>

Widget parent; /* assume this is an Xm Form, already created */
Widget plotter1, plotter2;
Arg args[10];
int i;

/* We create the first plotter with the constructor function. */
i = 0;
XtSetArg(args[i], XmNtitle, "Plotter #1"); i++;
XtSetArg(args[i], XmNtopAttachment, XmATTACH_FORM); i++;
XtSetArg(args[i], XmNleftAttachment, XmATTACH_FORM); i++;
XtSetArg(args[i], XmNrightAttachment, XmATTACH_FORM); i++;
XtSetArg(args[i], XmNbottomAttachment, XmATTACH_POSITION); i++;
XtSetArg(args[i], XmNbottomPosition, 50); i++;
plotter1 = XiCreatePlotter(parent, "top_plotter", args, i);

/* We create the second plotter with XtVaCreateWidget */
plotter2 = XtVaCreateWidget("bottom_plotter",      /* name */
                            xiPlotterWidgetClass, /* class */
                            parent,                /* parent */
                            XmNtitle, "Plotter #2",
                            XmNleftAttachment, XmATTACH_FORM,
                            XmNrightAttachment, XmATTACH_FORM,
                            XmNbottomAttachment, XmATTACH_FORM,
                            XmNtopAttachment, XmATTACH_POSITION,
                            XmNtopPosition, 51,
                            NULL);
XtManageChild(plotter1);
XtManageChild(plotter2);
```

5.2 Plotter Resources

The Plotter widget has resources that control its title, legend, plotting area, user interaction callbacks, and other miscellaneous features. Also, as a constraint widget, the Plotter provides constraint resources to each of its children. These resources are summarized in Table 5-1, and are documented in detail in the subsections below.

Table 5-1. Plotter Resources

Name	Class	Type	Default	CSG	Pg
Colors					
XmNforeground	XmCForeground	Pixel	black	CSG	56
XmNbackground	XmCBackground	Pixel	white	CSG	56
XmNplotAreaColor	XmCBackground	Pixel	XmNbackground[a]	CSG	56
XmNbackdrop	XmCBackdrop	Widget	NULL	SG	57
Fonts					
XmNfontList	XmCFontList	XmFontList	see below	CG	57
XmNtitleFontList	XmCFontList	XmFontList	XmNfontList[c]	CSG	58
XmNlegendTitleFontList	XmCFontList	XmFontList	XmNfontList[c]	CSG	58
XmNlegendFontList	XmCFontList	XmFontList	XmNfontList[c]	CSG	58
XmNpsFontList	XmCPSFontList	XiPSFontList	NULL	CG	58
XmNtitlePSFontList	XmCPSFontList	XiPSFontList	XmNpsFontList[d]	CG	58
XmNlegendTitlePSFontList	XmCPSFontList	XiPSFontList	XmNpsFontList[d]	CG	59
XmNlegendPSFontList	XmCPSFontList	XiPSFontList	XmNpsFontList[d]	CG	59
Axes					
XmNxAxis[e]	XmCXAxis	Widget	NULL	G	59
XmNyAxis[e]	XmCYAxis	Widget	NULL[f]	G	59
XmNyAxis2[e]	XmCYAxis	Widget	NULL	G	59
XmNsecondYAxis	XmCSecondYAxis	Boolean	FALSE	CSG	59
Title					
XmNshowTitle	XmCShowTitle	Boolean	TRUE	CSG	61
XmNtitleString	XmCTitleString	XmString	NULL	CSG	60
XmNtitle	XmCTitle	String	NULL	CSG	60
XmNtitleSize	XmCFontSize	Dimension	18	CSG	60
XmNtitleStyle	XmCFontStyle	String	NULL	CSG	60
XmNtitleColor	XmCForeground	Pixel	XmNforeground[b]	CSG	61
Legend					
XmNshowLegend	XmCShowLegend	Boolean	TRUE	CSG	61
XmNlegendTitleString	XmCLegendTitleString	XmString	NULL[f]	CSG	62
XmNlegendTitle	XmCLegendTitle	String	"Legend"	CSG	62
XmNlegendTitleSize	XmCFontSize	Dimension	14	CSG	63
XmNlegendTitleStyle	XmCFontStyle	String	NULL	CSG	63
XmNlegendTitleColor	XmCForeground	Pixel	XmNforeground[b]	CSG	63

XmNlegendSize	XmCFontSize	Dimension	12	CSG	63
XmNlegendStyle	XmCFontStyle	String	NULL	CSG	63
XmNlegendColor	XmCForeground	Pixel	XmNforeground[b]	CSG	63
XmNlegendBackground	XmCBackground	Pixel	XmNbackground[a]	CSG	63
XmNlegendSelectForeground	XmCBackground	Pixel	XmNbackground[a]	CSG	63
XmNlegendSelectBackground	XmCForeground	Pixel	XmNforeground[b]	CSG	64
XmNlegendSpacing	XmCMargin	Dimension	4	CSG	64
XmNlegendIconWidth	XmCLegendIconWidth	Dimension	20	CSG	64
XmNlegendUnderline	XmCLegendUnderline	Boolean	TRUE	CSG	64
XmNlegendBox	XmCLegendBox	Boolean	TRUE	CSG	64
XmNlegendBoxWidth	XmCLineWidth	Dimension	2	CSG	64
XmNlegendBoxColor	XmCForeground	Pixel	XmNforeground[b]	CSG	64
Layout					
XmNmarginWidth	XmCMargin	Dimension	10	CSG	65
XmNmarginHeight	XmCMargin	Dimension	10	CSG	65
XmNtitleMinHeight	XmCTitleMinHeight	Dimension	5	CSG	65
XmNlegendMinWidth	XmCLegendMinWidth	Dimension	5	CSG	65
XmNlegendMargin	XmCLegendMargin	Dimension	10	CSG	65
Double Buffering					
XmNdoubleBuffer	XmCDoubleBuffer	Boolean	FALSE	CSG	66
XmNdoubleBufferFrequencyHint	XmCDoubleBufferFrequencyHint	DoubleBufferFrequencyHint	Frequent	CSG	66
XmNbackingStore	XmCBackingStore	BackingStore	FALSE	CSG	67
User Interaction					
XmNallowCrosshair	XmCAllowCrosshair	Boolean	TRUE	CSG	234
XmNcrosshairLineWidth	XmCLineWidth	Dimension	0	CSG	234
XmNcrosshairColor	XmCForeground	Pixel	XmNforeground[b]	CSG	234
XmNpermanentCrosshair	XmCPermanentCrosshair	Boolean	FALSE	CSG	235
XmNcrosshairMotionOnly	XmCCrosshairMotionOnly	Boolean	FALSE	CSG	
XmNallowDrag	XmCAllowDrag	Boolean	TRUE	CSG	238
XmNdragThreshold	XmCDragThreshold	Dimension	4	CSG	231
XmNdragLineWidth	XmCLineWidth	Dimension	0	CSG	238
XmNdragColor	XmCForeground	Pixel	XmNforeground[b]	CSG	238
XmNallowPan	XmCAllowPan	Boolean	TRUE	CSG	241
XmNpanLineWidth	XmCLineWidth	Dimension	0	CSG	241
XmNpanColor	XmCForeground	Pixel	XmNforeground[b]	CSG	241

XmNallowMultipleSelect	XmCAllowMultipleSelect	Boolean	TRUE	CSG	229
Callbacks					
XmNactivateCallback	XmCCallback	Callback	NULL	C	229
XmNclickCallback	XmCCallback	Callback	NULL	C	231
XmNdoubleClickCallback	XmCCallback	Callback	NULL	C	231
XmNdragCallback	XmCCallback	Callback	NULL	C	237
XmNdragMotionCallback	XmCCallback	Callback	NULL	C	237
XmNeditCallback	XmCCallback	Callback	NULL	C	243
XmNleaveCallback	XmCCallback	Callback	NULL	C	234
XmNmotionCallback	XmCCallback	Callback	NULL	C	233
XmNpanCallback	XmCCallback	Callback	NULL	C	240
XmNpanMotionCallback	XmCCallback	Callback	NULL	C	240
XmNselectCallback	XmCCallback	Callback	NULL	C	228
Cursors					
XmNplotterCursor	XmCCursor	ColoredCursor	None	CSG	68
XmNclickCursor	XmCCursor	ColoredCursor	crosshair	CSG	68
XmNpanCursor	XmCCursor	ColoredCursor	fleur	CSG	241
XmNtopLeftCursor	XmCCursor	ColoredCursor	ul_angle	CSG	68
XmNtopRightCursor	XmCCursor	ColoredCursor	ur_angle	CSG	68
XmNbottomLeftCursor	XmCCursor	ColoredCursor	ll_angle	CSG	68
XmNbottomRightCursor	XmCCursor	ColoredCursor	lr_angle	CSG	68
Printing					
XmNforceColorPrinting	XmCForceColorPrinting	Boolean	FALSE	CSG	
XmNcustomPSPrefix	XmCCustomPSPrefix	String	NULL	CSG	69
XmNcustomPSSuffix	XmCCustomPSSuffix	String	NULL	CSG	69
Constraint Resources					
XmNbackgroundPlot	XmCBackgroundPlot	Boolean	FALSE	CSG	70
XmNlegendString	XmCLegendString	XmString	NULL[f]	CSG	70
XmNlegendName	XmCLegendName	String	NULL	CSG	70
XmNdrawingOrder	XmCDrawingOrder	Short	1	CSG	70
XmNselected	XmCSelected	Boolean	FALSE	CSG	71

[a] *Inherited from the XmNbackground resource.*

[b] *Inherited from the Xmforeground resource.*

[c] *Inherited from the XmNfontList resource.*

[d] *Inherited from the XmNpsFontList resource*

[e] *Plotter axes are implemented as independent Axis objects, each with a full set of its own resources.*

[f] *When an XmString resource is unset, its value is created from the corresponding String resource, using a font tag derived from the corresponding style and size resources.*

5.2.1 Default Colors and Background Resources

The resources documented in this section control the foreground and background colors for the plotter. They are not often actually used directly by the Plotter widget, but instead serve as default values for other color resources--resources that control the colors of the individual features of the Plotter.

XmNforeground

When a Plotter widget is created, the value of this resource is used as the default for all the other foreground resources of the widget, but it is never actually used itself. You can set this resource when you create the widget (or specify it in a resource file) as a simple way to specify a foreground for all the various parts of the widget. Setting the resource after the widget has been created has no effect, however, since its value is used only for inheritance.

XmNbackground

This resource specifies the background color for the Plotter. If the XmNplotAreaColor and the XmNlegendBackground resources are not specified when the widget is created, then this resource is used for the plot area and legend backgrounds as well. Note that this is not actually a Plotter resource; it is inherited from the Core widget class. If you change this resource, the change is not propagated on to the plotting area and the legend. If these regions inherited their color from the XmNbackground resource, then you will probably want to change their colors explicitly if you ever change XmNbackground.

There is another difficulty with changing the XmNbackground resource. When this color changes, the change is not passed on to the Axis children of the Plotter. The Y axis displays rotated text in a Pixmap and the background color of this Pixmap does not change when the XmNbackground resource does. A workaround is to set the Y axis title resource to NULL and then restore it to its original value. This will cause the title text to be re-rotated, and the Pixmap recreated with the correct background.

XmNplotAreaColor

This resource specifies the background color of the plotting area (the region between the axes). If no value is specified for this resource when the widget is created, its value is inherited from the XmNbackground resource of the widget. On color displays, you can achieve visually appealing effects using different colors for XmNbackground and XmNplotAreaColor. Some plot types inherit this background color to use as the background color for markers and fill patterns. If you change XmNplotAreaColor, the change will not automatically propagate to these children. To be safe, you should only change this resource when there are no plots being displayed in the plotter.

XmNbackdrop

This resource specifies a Plot widget child of the Plotter which will draw the background for the Plotter widget. You can use this resource with special plot types to achieve special visual effects for presentation graphics. Currently the only type of plot widget suitable for use as a "backdrop plot" is the FadePlot widget. See Chapter 14, *The FadePlot Widget* , for more information about backdrop plots and an example of using this XmNbackdrop resource.

5.2.2 Fonts

The resources described in this section specify the fonts to be used to display text in the Plotter widget, and the corresponding fonts to be used in PostScript hardcopy. Four of these resources are of the Motif XmFontList type and specify one or more fonts to be used to display XmStrings in a particular part of the Plotter widget. Note that most of the font list resources inherit their default value from the XmNfontList resource, so that if you specify an adequate number of fonts on this one font list, you need not specify any of the other font list resources.

Each XmFontList resource has a corresponding XiPSFontList resource, which specifies the PostScript equivalents of each of the X fonts listed in the XmFontList. Note that the Plotter widget can automatically determine the PostScript equivalents for the standard Adobe X fonts--i.e. the Adobe Times, Helvetica, Courier, New Century Schoolbook, and Symbol font families--so if you are using only fonts from these families, you will not need to specify any of the XiPSFontList resources. Note that the default values for XiPSFontList resources are inherited in the same way that the XmFontList resource defaults are.

See Section 3.3, *Fonts and DisplayedText*, for more information on fonts and text display in the XiPlotter widget.

XmNfontList

When the Plotter widget is created, the value of this resource is used as the default for all the other XmFontList resources in the widget and in its children. Like the XmNforeground resource, however, it is never actually used itself. XmNfontList is used only for inheritance when the widget is created; changing its value with XtSetValues() will have no effect. The default value of this default font list is the following:

```
-*-helvetica-medium-r-*-*-*-120-*-*-*-*-*-*=12,
-*-helvetica-medium-r-*-*-*-100-*-*-*-*-*-*=10,
-*-helvetica-medium-r-*-*-*-140-*-*-*-*-*-*=14,
-*-helvetica-medium-r-*-*-*-180-*-*-*-*-*-*=18
```

These fonts and their font tags are chosen to work with the default values of the various font size and font style resources throughout the Plotter widget and its children. See Section 3.3 for a further discussion. When working with the Plotter widget, you may choose to specify a single comprehensive font list, like the default above, that contains the fonts used throughout the plotter, or you may prefer to specify separate, individual fonts on the various font list resources described below.

XmNtitleFontList

The font or list of fonts to be used to display the title of the Plotter widget. If no value is specified for this resource, its default value is inherited from the XmNfontList resource.

XmNlegendTitleFontList

The font or list of fonts to be used to display the title of the legend in the Plotter widget. If no value is specified for this resource, its default value is inherited from the XmNfontList resource.

XmNlegendFontList

The font or list of fonts to be used to display the items in the legend of the Plotter widget. If no value is specified for this resource, its default value is inherited from the XmNfontList resource.

XmNpsFontList

This resource is an XiPSFontList that is not used directly by the Plotter widget itself, but instead serves as the default value for all the other XiPSFontList resources in the Plotter widget and its Plot children. An XiPSFontList specifies PostScript fonts that correspond to each of the X fonts in an XmFontList. No XiPSFontList is required when using any of the standard Adobe X fonts--the Times, Helvetica, Courier, New Century Schoolbook, and Symbol font families--and so the default for this resource is NULL.

Note that while the name of this resource is XmNpsFontList, its class name is XmCPSFontList, instead of the expected XmCPsFontList--this minor departure from the usual resource class name capitalization conventions is for legibility.

See Section 3.3 for more information on specifying XiPSFontList resources from C and from a resource file.

XmNtitlePSFontList

The list of PostScript fonts to be used to display the title of the Plotter widget in hardcopy. If no value is specified for this resource, its default value is inherited from the XmNpsFontList resource.

XmNlegendTitlePSFontList

The list of PostScript fonts to be used to display the title of the legend of the Plotter widget in hardcopy. If no value is specified for this resource, its default value is inherited from the XmNpsFontList resource.

XmNlegendPSFontList

The list of PostScript fonts to be used to display the items in the legend of the Plotter widget in hardcopy. If no value is specified for this resource, its default value is inherited from the XmNpsFontList resource.

5.2.3 The Axes

The X and Y axes and the optional secondary Y axis are probably the most important parts of the plotter window. They are also the most complex--each axis has over 60 resources that control its bounds, labeling, and overall appearance. Because there are so many resources, the axes are implemented as a separate widget type. When you create an XiPlotter widget, it automatically creates two XiAxis children--each with its independent set of resources--for the X and Y axes, and if XmNsecondYAxis is True, it creates a third XiAxis widget for the secondary Y axis. The XiAxis widget and all of its resources are documented in Chapter 6, *The Axis Widget*.

The Plotter widget resources that are relevant to the axes are the following:

XmNxAxis, XmNyAxis, XmNyAxis2

These are read-only resources that you can use to obtain a Widget pointer for each of the axis widgets. With this pointer you can set XiAxis resources on the axes. See below, however: there are also convenience functions which will return these axis Widget pointers.

XmNsecondYAxis

This is a Boolean resource that specifies whether or not the Plotter should display a secondary Y axis on the right-hand side of the plotting area. A secondary Y axis is not fully independent: its minimum and maximum bounds must be linearly proportional to the bounds of the primary Y axis, and the secondary axis bounds will automatically change when the primary axis bounds do. Note that setting XmNsecondYAxis to True creates a new Axis widget, and setting the resource to False destroys it. If you want to simply hide the secondary Y axis temporarily, it is better to simply unmanage it, instead of destroying it and then recreating it later. See Chapter 6 for more information about secondary axes.

The Plotter widget provides three functions, declared in <Xi/Plotter.h>, for obtaining the axes of a Plotter. XiPlotterXAxis(), XiPlotterYAxis(), and XiPlotterYAxis2() each take a Plotter widget as their single argument and return the X axis, the Y axis, and the secondary Y axis widgets of that Plotter. You can also obtain these resources by querying the XmNxAxis, XmNyAxis, and XmNyAxis2 resources. (These resources are read-only; you should not set them.)

Once you have obtained pointers to the Axis widgets by either of these methods, you can set any of the Axis resources, as you would for any other widget. You can also manage and unmanage the axes. An unmanaged axis will not appear in the plotter; this can be useful when displaying pie charts, for example.

5.2.4 The Title

The Plotter widget optionally displays a title centered above the plotting area. The following resources control the text and the appearance of that title.

XmNtitleString

The XmString to be displayed as the title of the Plotter. If no XmString is specified for this resource, then one will be created using the XmNtitle string, and a font list tag constructed from the XmNtitleStyle and XmNtitleSize resources, as described in Section 3.3, *Fonts and Displayed Text.* Setting this resource will override any value previously created from the XmNtitle string. If you query this resource, the Plotter returns its private, internal copy of the XmString, which you must not modify or free.

XmNtitle

This is the string to appear as the title of the Plotter, if no XmNtitleString is specified. Note that this is an ordinary NULL-terminated character string, not a Motif XmString. If you set this resource, the string will be converted to an XmString using a font list tag constructed from the XmNtitleStyle and XmNtitleSize resources, and this newly created XmString will become the new value of the XmNtitleString resource. See Section 3.3 for details. When you set XmNtitle, the Plotter widget makes an internal copy of it, so you are free to modify or change your copy of the string after setting it. If you query this resource, the returned string is the widget's internal copy, which you must not modify or free.

XmNtitleSize

The point size of the Plotter title text. This value is used in the font list tag of the XmNtitle, if no XmNtitleString is specified. The default value is 18.

XmNtitleStyle

The typeface for the Plotter title text. This resource is used in the font list tag of the XmNtitle, if no XmNtitleString is specified. XmNtitleStyle is a string resource, and the default value is NULL. If you set this resource, you will typically set it to strings such as "bold" and "italic", depending on the font tags in your XmNtitleFontList. When you set this resource, the string value is copied, and when you query it, you must not modify or free the returned value.

XmNtitleColor

This is the color the Plotter title is to be displayed in. If no value is specified for this resource when the widget is created, then the value of the XmNforeground resource will be used. (Once the widget is created, however, changing the XmNforeground resource will not change the color of the title.)

XmNshowTitle

A Boolean resource that specifies whether the title should be displayed. The default is True which specifies that the title should be displayed, if any is specified. If set to False, then no title will be displayed, even if one is specified.

5.2.5 The Legend

The Plotter widget will optionally display a "legend" or "key" to the right of the plotting area. When there is more than one Plot displayed in the Plotter, this legend provides a name for each plot so that the user can identify them. The Plotter legend is also sensitive to mouse clicks, and, like the Motif and Athena List widgets, it allows users to select any of the displayed items.

There are many resources that control the appearance of the legend; they are described below. There are also two Plotter constraint resources used by the legend. The XmNlegendName constraint lets you specify the name to appear in the legend for a particular plot, and the XmNselected constraint lets you specify whether that name should appear highlighted in the legend. These constraint resources are documented in Section 5.2.11, *Constraint Resources*.

The resources that control the appearance and layout of the Plotter legend are described below. Figure 5-1 shows a diagram of a plotter legend, which is useful in figuring out what each of these resources do.

XmNshowLegend

A Boolean resource that specifies whether the legend should be displayed. The default is True. If False, then the legend will not be displayed in the Plotter.

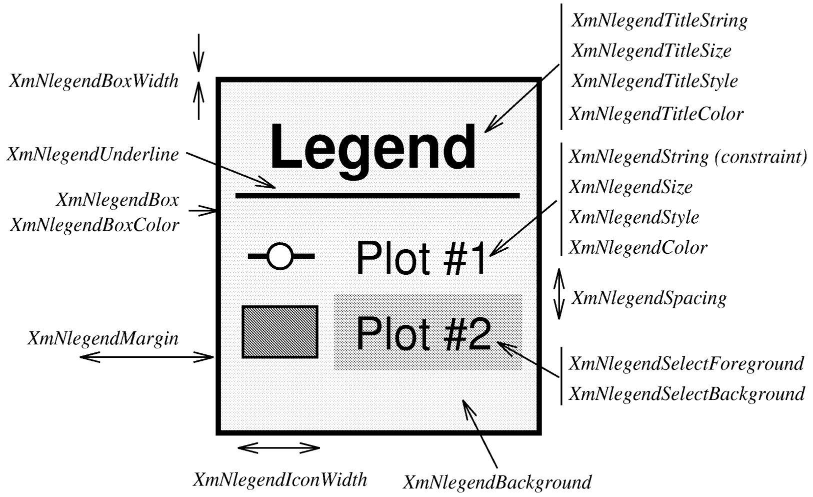


Figure 5-1. The XiPlotter legend

XmNlegendTitleString

The XmString to be displayed as the title of the Plotter legend. If this resource is not specified, then an appropriate XmString will be created using the XmNlegendTitle string and a font list tag constructed from the XmNlegendTitleStyle and XmNlegendTitleSize resources, as Text. Setting this resource will override any value previously created from the XmNlegendTitle string. If you query this resource, the Plotter returns its private, internal copy of the XmString, which you must not modify or free.

XmNlegendTitle

The string that appears at the top of the legend, if no XmNlegendTitleString is specified. This resource is an ordinary NULL-terminated character string, not an XmString. If the XmNlegendTitleString is not specified, then this string will be converted to an XmString, using a font list tag constructed from the XmNlegendTitleStyle and XmNlegendTitleSize resources, and this newly created XmString will become the value of the XmNlegendTitleString resource. See Section 3.3 for details.

The default value of this resource is the string "Legend", but you might want to change it to "Key", or some other value. When you set this resource, the Plotter makes its own private copy of the string, and the value is used to create an

XmString that overrides any previously specified value for XmNlegendTitleString. When you query this resource, the string returned is a private, internal copy, which you must not modify or free.

XmNlegendTitleSize

A point size to use in the font list tag of the XmNlegendTitle, if no XmNlegendTitleString is specified. The default is 14.

XmNlegendTitleStyle

A typeface name to use in the font list tag of the XmNlegendTitle, if no XmNlegendTitleString is specified. The default is NULL. When you set this resource, the string value is copied, and when you query it, you must not modify or free the returned value.

XmNlegendTitleColor

The color of the legend title. If this value is not set when the widget is created, it will inherit the value of the XmNforeground resource.

XmNlegendSize

The point size of the font used for entries in the legend. If no XmNlegendString constraint is specified for a plot in the Plotter, then this value is used in the font list tag for the XmNlegendName constraint of that item. The default is 12. Note that changing the value of this resource will not cause the size of already-existing legend items to change. See Section 3.3 for more information.

XmNlegendStyle

The typeface for entries in the legend. If no XmNlegendString constraint is specified for a plot in the Plotter, then this value is used in the font list tag for the XmNlegendName constraint of that item. The default is NULL. Note that changing the value of this resource will not cause the typeface of already-existing legend items to change. See Section 3.3 for more information. When you set this resource, the string value is copied, and when you query it, you must not modify or free the returned value.

XmNlegendColor

The color of the text for each entry in the legend. If this resource is not set when the widget is created, its value is inherited from the XmNforeground resource.

XmNlegendBackground

The background color for the legend. If this resource is not specified when the widget is created, its value is inherited from the XmNbackground resource.

XmNlegendSelectForeground

The foreground color of any selected items in the legend. If this resource is not set when the widget is created, its value is inherited from the XmNbackground resource.

XmNlegendSelectBackground

The background color of any selected items in the legend. If this resource is not set when the widget is created, its value is inherited from the XmNforeground resource, which *will* use selected items to be displayed in "reverse video".

XmNlegendSpacing

The number of blank pixels between items in the legend. The default is 4.

XmNlegendIconWidth

The width, in pixels of the "icon" for each legend item. The "icon" is the section of the legend that graphically identifies the plot. The default value is 20.

XmNlegendBox

A Boolean resource that specifies whether a box should be drawn around the legend. The default is True. If False, then no box will be drawn.

XmNlegendUnderline

A Boolean resource that specifies whether the legend title should be underlined. The default is True. If False, then no line will be drawn.

XmNlegendBoxWidth

The line width to be used to draw the box around the legend and the line beneath the legend title. The default is 2.

XmNlegendBoxColor

The color to be used for the legend box and underlining. If this resource is not set when the widget is created, then it inherits its value from the XmNforeground resource.

5.2.6 Plotter Layout

The Plotter widget automatically arranges its title, legend, axes and plotting area in very reasonable default places. The resources documented in this section allow you an extra degree of control over the size of margins in the Plotter widget and the size of the title and legend within the Plotter. The related Axis widget resource, XmNaxisWidth can be used to control the positions of the axes.

XmNmarginWidth

The width, in pixels, of the margin at the left and right sides of the plotter. The default is 10.

XmNmarginHeight

The height, in pixels, of the margin at the top and bottom of the plotter. The default is 10.

XmNlegendMargin

The number of pixels between the left edge of the legend and the right edge of the plotting area. This resource does not take the width of the lines used to frame the legend and plotting area into account; if you use wide lines, the number of blank pixels will appear smaller than specified. The default value is 10.

XmNlegendMinWidth

The minimum number of pixels to be allocated for the legend at the right of the plotter. The legend will always be at least this many pixels wide, but will be wider if needed. This resource is useful if you have two plotter widgets displayed one above the other, and want to force their legends to be the same width so that their plotting areas are aligned. The default value is 5, which serves as an internal margin when no legend is displayed. If no legend is displayed, and there is a long label at the end of the X axis, you may have to adjust this minimum width to make room for the axis label. See also XmNtitleMinHeight and the Axis resource XmNaxisWidth.

XmNtitleMinHeight

This is the minimum height of the title in pixels. The plotter always allocates at least this many pixels of its window to display the title. This can be useful when you want to align the titles of adjacent Plotter widgets, even when those widgets use different title fonts--you can simply set the XmNtitleMinHeight resource of both to the height of the larger font.

The Plotter never chops off the top or bottom of the title--if the title font is taller than this resource, the Plotter will ignore this resource and allocated as much of its window as it needs for the title. The default value for this resource is 5 pixels; is this smaller than any actual title that would be specified, but serves as an internal margin when no title is specified or when XmNshowTitle is False. (This internal margin is important to prevent the topmost label on the Y axis from extending out of the window when no title is displayed in the Plotter. If you are using a larger-than-usual font for the labels on the Y axis, you may have to use a larger value for XmNtitleMinHeight. See also XmNlegendMinWidth and the Axis resource XmNaxisWidth.

5.2.7 Double Buffering and Backing Store

The Plotter widget has the ability to use *double buffering* if you request it. When double buffering is turned on, the Plotter always redraws itself into an off-screen "buffer", and then when the redraw is complete, it copies the fully-drawn contents of the buffer into the on-screen Plotter window. In practice, this makes redraws very crisp and flicker-free. Double-buffering is not turned on by default, because it requires a fair bit of memory in the X server to implement. If your program will be displaying data that changes frequently, you will almost always want to turn double buffering on--it will make the resulting display much nicer. The Plotter widget implements double-buffering with the MBX multi-buffering extension to X, if the X server it is talking to supports it. If not, it simply performs double-buffering on its own by allocating its own off-screen bitmap.

The plotter widget also supports *backing store*, which, like double buffering is a way of decreasing the amount of flickering that the user sees on the screen. When backing store is turned on, the Plotter widget will never have to redraw itself when it is covered and uncovered--i.e. the X server remembers the contents of the Plotter window, even when that window is obscured by other windows. Backing store can be useful when a Plotter is displaying a particularly complex set of plots that are time consuming to redraw. Bear in mind, though, that backing store takes up a lot of memory in the server, so it is not turned on by default and it should only be used in special circumstances.

The resources that control double buffering and backing store are the following:

XmNdoubleBuffer

A Boolean that specifies whether double buffering is turned on. Setting this resource to True turns on double buffering. The default is False.

XmNdoubleBufferFrequencyHint

When double buffering is on, and the X server supports the MBX multi-buffering extension, this resource is a hint to the server that tells it how often you think that the Plotter window will be redrawn. The legal values are defined in <X11/extensions/multibuf.h>: MultibufferUpdateHintFrequent, MultibufferUpdateHintIntermittent, and MultibufferUpdateHintStatic. When memory is scarce, the X server may use this hint to decide which window have the greatest need for off-screen buffers, and deny double-buffering to those windows that will not be updated frequently. Many X servers will simply ignore this resource.

The Plotter widget registers a type converter so that you can specify a value for this resource from a resource file. The converter recognizes the values "Frequent", "Intermittent", and "Static".

XmNbackingStore

This resource specifies whether backing store should be on, and if so, the type of backing store required by the widget. There are three possible values, which are defined in <X11/X.h>: NotUseful, WhenMapped, and Always. The default is NotUseful, which means that the widget does not request any backing store from the server; whenever the plotter is obscured and then uncovered it will have to redraw itself. The value WhenMapped means that the widget requests backing store from the server, except when it is iconified. This means that the widget will generally never have to redraw itself when covered and uncovered. The value Always means that backing store is requested, even when the widget is iconified. This means that the widget does not have to redraw itself even when iconified. Remember that X servers are not required to implement backing store, nor to provide it for every window that requests it, so this resource must be considered only a hint to the server.

The Plotter registers a type converter for this resource, so that you can specify values from a resource file. The converter recognizes the strings "NotUseful", "WhenMapped", and "Always".

5.2.8 Interaction Resources and Callbacks

There are a number of Plotter resources that control user interaction with the Plotter. XmNallowDrag, for example specifies whether the plotter will display a rubber-banded rectangle when the user drags the mouse in the plotting area, and XmNdragColor specifies the color of that rectangle. The topic of user interaction also involves callbacks, translations, and actions. These subjects are important enough to warrant a chapter of their own. See Chapter 16, *User Interaction with the Plotter* for documentation of these topics and a complete explanation of all of the Plotter interaction resources.

5.2.9 Cursor Resources

The Plotter widget uses a number of different pointer cursors in different situations. The resources described in this section specify the cursors that are to be displayed. The XmNplotterCursor resource specifies the default cursor to be displayed. The other resources specify special cursors to be displayed during user interaction, and while the cursor resources are documented here, the user interactions they are used for are described in more detail in Chapter 16, *User Interaction with the Plotter.*

You can set these resources to any cursor (a value of type Cursor in Xlib) you create. You can also specify cursors by name in a resource file. The Plotter widget registers an enhanced String-to-Cursor resource converter that also allows you to specify foreground and background colors for your cursors. To use it, simply follow the cursor name (the cursor must be from the standard cursor font) with an the desired foreground and background color names. If you do not specify colors, black will be used for the foreground, and the background will be inherited from the Plotter XmNbackground resource. See Section 3.9, *Cursors*, for more information about cursors and about the Plotter's enhanced cursor converter.

The Plotter cursor resources are the following:

XmNplotterCursor

The cursor to display when the pointer is inside the Plotter widget, there are no crosshairs displayed, and the user is not clicking, dragging, or panning. The default is None, which specifies that the cursor should be inherited from the Plotter widget's parent. Generally, a value of None will result in the default arrow cursor being used.

XmNclickCursor

The cursor to be displayed while the user has the mouse button down during a click. The default is the cursor named "crosshair" from the standard cursor font.

XmNcrosshairCursor

The pointer cursor to be displayed while the crosshairs are displayed. The default value of this resource is the cursor named "crosshair".

XmNtopLeftCursor, XmNtopRightCursor, XmNbottomLeftCursor, XmNbottomRightCursor

These resources specify the four cursors to be used when the user drags out a rectangle. For example, the XmNbottomRightCursor will be displayed when the user selects a rectangle by clicking at the upper left and dragging diagonally down and to the right--the bottom right corner of the "rubberbanded" rectangle will follow the pointer. The default values for these cursor resources are angles that match the appropriate corner of the rectangle. If you change one of these resources, you will probably want to change them all to provide a matching set.

XmNpanCursor

The cursor to display when the user pans the widget. The default is the cursor named "fleur" in the standard cursor font. (This cursor displays arrows pointing in each of the four cardinal directions.)

5.2.10 Printing Resources

The resources described in this section control the generation of PostScript hardcopy when the plotter is printed with XiPlotterPrint(). See Section 5.4, *Printing the Plotter Widget*, for more information about creating PostScript hardcopy of the Plotter widget.

XmNforceColorOutput

The Plotter widget always produces color PostScript output. When color PostScript is printed on a monochrome device (like most laser printers) all colors are converted to a shade of gray. In many cases, these shades of gray distract from the data being displayed, especially when used as a background color. Thus, by default, the Plotter outputs PostScript which checks whether it is being printed on a monochrome device, and if so, always produces a white background and uses a black foreground for the axes, for text, for the lines of line plots, and so on. In this default printing mode, only filled areas, such as the bars in bar charts and the wedges in pie charts will be printed as shades of gray. This default behavior generally produces crisper, more legible output on monochrome devices, and still allows full-color printing for color printers on on-screen previewers. Sometimes, however, you will want to override this default behavior and produce monochrome printouts that use shades of gray for the entire widget. You can do this by setting XmNforceColorOutput to True.

XmNcustomPSPrefix, XmNcustomPSSuffix

These two resources specify strings that will be inserted at the beginning and end of all PostScript output. Using them requires a thorough understanding of the PostScript language, and also of the structure of the PostScript page descriptions that the Plotter outputs. If you knew that your output were to be printed on a 300dpi laser printer, for example, you might use XmNcustomPSPrefix, for example, to output a PostScript setscreen operator that would produce shades of gray with a smaller, less grainy stipple, at the price of reducing the number of shades of gray available.

5.2.11 Constraint Resources

The XiPlotter is a subclass of the Xt Constraint widget class. This means that it is a "constraint widget", and, like the Motif XmForm widget, for example, it provides "constraint resources" to each of its children. A "constraint resource" (or just "constraint") is a special resource provided and handled by the constraint widget which you set on any of the children of that widget. The Plotter widget provides four such constraint resources, which you can set on any of the Plot children that appear in that widget. These constraints are the following:

XmNbackgroundPlot

This constraint resource is a Boolean value. If True, it indicates that this Plot should be drawn "in the background"--i.e. beneath any other plots, and also beneath the grid lines of the Plotter. This constraint can be used with any Plot widget, but is currently most useful with the FadePlot widget (and perhaps the ImagePlot widget). Do not confuse this constraint resource, which specifies that any given plot should be drawn in the background of the plotting area, with the Plotter XmNbackdrop resource, which is not a constraint, and which specifies a single Plot to be drawn as the background of the entire Plotter window.Chapter 14, *The FadePlot Widget,* shows an example of using a FadePlot widget with the XmNbackdrop resource; that example is easily adapted to use the XmNbackgroundPlot constraint instead.

XmNlegendString

The XmString to be displayed in the legend for this plot. If no XmString is specified for this resource, then one will be created using the XmNlegendName resource (if it is specified) and a font list tag constructed from the XmNlegendStyle and XmNlegendSize resources, as described in Section 3.3, *Fonts and Displayed Text*. Note that all entries in the Plotter legend will be as tall as the tallest one. Thus you should generally use the same fonts for all legend entries, and make them the same number of lines high. Setting this resource will override any value previously created from the XmNlegendName string. If you query this resource, the Plotter returns its private, internal copy of the XmString, which you must not modify or free.

XmNlegendName

This is the string that is to appear in the legend for this plot, if no XmNlegendString is specified. Note that this is an ordinary NULL-terminated character string, not a Motif XmString. The default is NULL which specifies that this plot should not have an entry in the legend. If you set this resource, the string will be converted to an XmString using a font list tag constructed from the XmNlegendStyle and XmNlegendSize resources, and this newly created XmString will become the new value of the XmNlegendString constraint. See Section 3.3 for details. The Plotter widget makes an internal copy of this resource whenever it is set on a Plot child, so once you have set the resource, you are free to modify or free your copy of the string. If you query this resource, however, you obtain the internal copy of the string, which you must not modify or free.

XmNdrawingOrder

This short int resource specifies the order in which plots are drawn in the Plotter, and the order that they appear in the legend. The default is -1, which is a special value that indicates that the plot should be drawn after all other plots in the Plotter. (This means that if you never set this resource, plots will be drawn and will appear in the legend in the order that they were created.) The order that plots are drawn in determines their "stacking order"-- a plot will overlap any plots that were drawn before it. You can specify any value for this constraint, but note that positions 0 and 1 are generally reserved for the X and Y axes, which are special Plot widgets themselves.

XmNselected

This is a Boolean resource that specifies whether the Plot will appear selected in the legend. The default is False. The user can select any Plot in the Plotter by clicking or shift-clicking on its entry in the legend. Typically, you will handle this selection through one of the Plotter callbacks, but you can also explicitly set or query the selection state of any Plot with this constraint resource.

5.3 Plotter Redisplay

Changing the resources of the Plotter widget or of its axes or Plot children will very often cause the plotter to redisplay itself. Similarly, creating, destroying, managing or unmanaging a new Plot child will do the same. If you will be making more than one change to the Plotter or its children at a time, this multiple redisplay is inefficient and can cause annoying flickering on the screen.

To prevent multiple redisplay, call the function XiPlotterDisableRedisplay() before beginning a series of changes to the Plotter or its children. When you have completed a batch of changes, call XiPlotterEnableRedisplay() and the Plotter will update itself with a single redraw. Each of these functions takes a single Plotter widget as its only argument, and has no return value.

```
void XiPlotterDisableRedisplay(Widget w);s
void XiPlotterEnableRedisplay(Widget w);
```

`w` A Plotter widget. When XiPlotterDisableRedisplay() is called for a widget, that widget will not redraw itself until the matching XiPlotterEnableRedisplay() is called.

XiPlotterDisableRedisplay() and XiPlotterEnableRedisplay() maintain a count of the number of times they have been called for any Plotter widget, so it is possible to nest calls to them--the Plotter will not redisplay itself until the "outermost" XiPlotterEnableRedisplay() is reached.

Whenever you change a resource or call a convenience function on the Plotter widget, its Axis widgets, or any of its children Plot widgets, the Plotter will generally have to do redisplay, rescale, or relayout itself. Changing the color of a Plot would force a redisplay, for example; adding a new plot might force a rescale, if the new plot caused the axes to autoscale; and changing the font in the Plotter title could cause a relayout if it were a different size than the previous font.

Because of the tremendous number of resources supported by the Plotter, Axis and the various Plot widgets, it is simply not possible to test the setting of all resources in all combinations. Although there are currently no known bugs of this sort, you may find instances when the Plotter widget incorrectly or incompletely updates itself after a resource has been set or a convenience function called. If you ever do notice this behavior, please report the bug, so we can fix it for the next release. Until the bug is fixed, you should be able to work around the problem with the function XiPlotterRelayout().

```
void XiPlotterRelayout(Widget plotter);
```

`plotter` A Plotter widget. When this function is called, the specified widget (and all of its children) will be completely refreshed: it will be laid out, rescaled, and redrawn from scratch.

Note that XiPlotterRelayout() is not intended for any regular use; it is provided with the Plotter library as a workaround, in case redisplay problems do arise with your applications.

5.4 Printing the Plotter Widget

You can generate a PostScript version of the Plotter widget and all the plots within it by calling the function XiPlotterPrint(). The signature for this function is shown in Figure 5-2.

XiPlotterPrint() generates an EncapsulatedPostScript file, which can be printed on its own or easily incorporated into other documents. The PostScript it generates will print in color on color devices, or in black and white and shades of gray on monochrome devices. See XmNforceColorOutput in Section 5.2.10, *Printing Resources*, for more information on how the Plotter handles color output.

5.4.1 Printout Size and Orientation

The width and height arguments to XiPlotterPrint() specify the size of the printed representation of the plotter. Both arguments are measured in PostScript points, which are exactly 1/72nd of an inch--to convert inches to points, multiply by 72. To convert centimeters to points, multiply by 28.35. Note that there are not x and y arguments to this function to specify where the Plotter should appear on the page. XiPlotterPrint() always centers its output on the page, which is generally appropriate if you will be printing out the page alone. If you will be incorporating the PostScript file into some other application, that application will let you place the image at the desired point on the page.

The landscape argument specifies whether the Plotter should be printed sideways on the page in "landscape mode". If you pass False, the printed image will be oriented normally in "portrait mode". If you pass True, it will be oriented sideways in landscape mode.

```
void XiPlotterPrint(Widget plotter, char *filename,
                    int width, int height,
                    double scale, int landscape);
```

`plotter` The XiPlotter widget that is to be printed in PostScript.

`filename`
The name of the PostScript file to generate.

`width, height`
The dimensions, in points, of the printed version of the Plotter. If either of these arguments is 0, its value will be computed from the value of the other argument and the on-screen aspect ratio of the Plotter widget. If both are zero, the printed version will occupy as much of the page as possible.

`scale` The "scale factor" for the printout. If the scale is 1.0, the printout will have features (font sizes, marker sizes, Axis tick lengths, etc.) that are approximately the same size as they appear on the screen. Smaller scales are appropriate when printing out a Plotter in a fairly small area. Most of the figures in this manual, for example, were generated with a scale of 0.5 or 0.75. Larger scales produce larger fonts, wider lines, and so on; they are appropriate when printing large plots for use on overhead transparencies, for example.

`landscape`
If this argument is True, the Plotter will be printed out in "landscape" mode (i.e. sideways on the page). If False, "portrait" mode will be used.

Figure 5-2. XiPlotterPrint()

The width and height arguments are both optional; you may pass 0 rather than specifying a value. Since either argument may be specified or not, there are four possible cases, each producing the output described below:

Both width and height specified.

The Plotter will be printed with the specified width and height, regardless of orientation or aspect ratio.

Width specified, height = 0.

The Plotter will be printed with the specified width. The height of the printed image will be computed by multiplying the width by the height:width ratio of the on-screen version of the widget.

Height specified, width = 0.

The Plotter will be printed with the specified height, and the width of the printout will be computed by multiplying that height by the width:height ratio of the on-screen version of the widget.

Width = 0, height = 0.

In this case, the Plotter will take up as much room as possible on the page (leaving at least a 3/4 inch margin all around), and will be printed with the same aspect ratio as the on-screen version of the widget. This is done by comparing the aspect ratio of the widget to the aspect ratio of the page--the page is assumed to be 8.5 inches wide and 11 inches high, but the aspect ratio depends on whether the Plotter is being printed in landscape or portrait mode. Depending on the results of this comparison, the width or the height of the image will be fixed at the width or height of the page (minus margins), and the other dimension of the image will be computed based on the widget aspect ratio as in the cases above.

5.4.2 Printout Scale

While the width and height arguments specify the absolute size of the printed image on the page, the scale argument specifies the relative size of the features within that image. If you specify a scale of 1.0, then the printed image will appear approximately as it does on the screen--the fonts will be the same height, the lines the same thickness, the Axis tick marks the same length, and so on. They will only be approximately the same, because of course the size on the screen depends on the rated resolution of your monitor, as well as the way you have tuned your monitor. The Plotter widget assumes that all X displays have a resolution of 90 pixels-per-inch. Since PostScript has 72 points-per-inch, the Plotter widget converts each on-screen pixel to 4/5ths of a point in PostScript. So a line drawn 5 pixels wide in X would be drawn 4 points wide in PostScript.

But the printed page is not the same as a computer monitor, and you do not always want PostScript output printed at the same scale that it appears on the screen. If you pass a scale argument of something other than 1.0, the output image will still have the width and height that you specified, but the details within the image will be larger or smaller. Smaller scale values are appropriate when you are printing a Plotter in a fairly small space, and don't want the fonts in the Plotter title, the legend, and the axes to take up all the space. Many of the figures in this book, for example, are 2 inches wide and 1.75 inches high, and were printed with a scale factor of 0.5. Because PostScript offers much higher resolution than X, the small scale factor allows these figures can be displayed on the page in much less room than would be required for screen display. Larger scale values can also be useful when printing out large versions of a Plotter. If you pass a width and height of 0, for example, and a scale of 2.0, then you will get an image that occupies most of the page, and has large fonts, fat lines, and so on. An image like this is ideal when generating overhead transparencies for presentations.

5.4.3 Spooling a PostScript File to a Printer

XiPlotterPrint() simply writes PostScript to a file; it does not actually spool anything to a printer. Sending PostScript code to a printer is a fairly system-dependent task and is left to the application. If you want to spool Plotter printouts from your application, you can follow a procedure like this:

1. Use tmpnam() (or mktemp() on BSD systems) to generate a unique temporary file name.
2. Call XiPlotterPrint() to generate the PostScript representation of the Plotter into that file.
3. Call system() to invoke the printer spooling command for your operating system. (Probably lpr.) For portability, you might want to allow this command to be configured at run time, or at least allow users to specify which printer they want (using the lpr -P option, for example).
4. Use unlink() to delete the temporary file.

The Axis Widget

When you create a Plotter, the widget automatically creates two XiAxis widgets to display its X and Y axes (and optionally a third XiAxis widget to display the secondary Y axis). The Axis and Plotter widgets are closely linked--neither will work without the other, and it is useful to think of the Axis widget not so much as an independent widget, but as a convenient placeholder for resources. There are 65 separate resources that control the appearance and numbering of each axis. Rather than add 195 resources to the Plotter widget--65 for each of three axes--it makes more sense to define the axes as separate objects each with an independent set of resources.

Axis Synopsis

Class Name: XiAxis

Class Hierarchy: RectObj → XiPlot → XiAxis

Header File: *<Xi/Axis.h>*

Class Pointer: `xiAxisWidgetClass`

Constructor: *none; automatically created by the Plotter*

This chapter documents the 62 public Axis resources that control the Axis, and also documents the functions that you can use to manipulate the axes of a Plotter. The first section explains how to obtain pointers to the Axis widgets from a Plotter widget, and the following two sections document the resources and functions that control Axis appearance and numbering.

The Axis is the most complicated of any of the widgets in the Plotter library. It has the following features:

- Each axis displays a title which can indicate what the axis is measuring, and what units is it using. A typical title for an X axis might be "Time (ms)", for example. For the Y axis, this title is displayed with text that has been rotated 90 degrees, so that it runs vertically along the axis. Resources specify the font and color of the title.
- The Axis displays major and minor tick marks (larger labeled marks, and smaller marks between the labels) along its length. The Axis will automatically (and intelligently) choose reasonable, round-number positions for the marks, or you can explicitly set the positions yourself. Resources let you specify the length, width, and color of the marks

- The Axis widget can display labels for each of the major tick marks it draws. By default, it will automatically label these marks with numbers. You can specify the way it formats these numbers, or you can provide your own array of strings for it to display.
- The Axis widget can autoscale itself so that all the data displayed in the Plotter will fall between the Axis minimum and maximum values. If you turn this autoscaling off, you can specify explicit minimum and maximum values.
- The Axis widget will draw lines perpendicular to itself across the plotting area. If both Axes do this, the lines form a grid, and provide a graph-paper appearance to the plotting region. The Axis can display both a "major" grid at only the labeled tick marks, and also a "sub-grid" at the smaller, unlabeled tick marks. Resources allow you to specify a color, width and line pattern for the lines of both grid types
- The Axis will draw a "frame" line parallel to itself on the opposite side of the plotting area. These frame lines combine with the Axis lines themselves to form a box around the plotting area. If the range of the axis includes the point 0.0, the Axis can also drawn an origin line perpendicular to itself at that point.
- The Axis supports logarithmic scaling, which allows you to easily produce semi-log or log-log plots. When logarithmic scaling is enabled, the Axis will automatically divide its range into decades, and will place its marks and grid lines appropriately. The Axis widget controls the scaling of Plot data, so plots will automatically be displayed correctly on the logarithmic axes.
- When the secondary Y axis is displayed, its minimum and maximum values are always linearly proportional to those of the primary Y axis. Two resources on the second Y axis specify how primary Y axis bounds are transformed into secondary Y axis bounds. When you set the bounds of the primary Y axis, the bounds of the secondary Y axis will be automatically updated. Similarly, when autoscaling changes the Y axis bounds, the secondary Y axis bounds change as well.

6.1 Querying the Axes of a Plotter

Since the Plotter creates its Axis widgets automatically, you'll need to obtain a pointer to them before you can manipulate the axes in any way.

```
Widget XiPlotterXAxis(Widget plotter);
Widget XiPlotterYAxis(Widget plotter);
Widget XiPlotterYAxis2(Widget plotter);
```

`plotter` An XiPlotter widget.

`Returns` The X or Y or secondary Y Axis widget created internally by the Plotter. If the *plotter* XmNsecondYAxis resource is False, then XiPlotterYAxis2()will return NULL.

You can get the X axis of a Plotter with XiPlotterXAxis(), and you can get the Y axis with XiPlotterYAxis(), as shown in Example 6-1.

Example 6-1. Querying the Axes of a Plotter widget

```
Widget plotter;
Widget xaxis, yaxis;

plotter = XtCreateManagedWidget("plotter", xiPlotterWidgetClass,
                                 parent, args, num_args);
xaxis = XiPlotterXAxis(plotter);
yaxis = XiPlotterYAxis(plotter);
```

If your application contains only one Plotter widget, you might query its axes right after you create it and store the returned pointers in global variables. If you are writing an application that uses multiple Plotters, you may find it easier to simply call XiPlotterXAxis() and XiPlotterYAxis() each time you need to obtain an Axis widget from a Plotter.

However you choose to obtain these Axis widgets, you can manipulate them as you would any other widget with XtSetValues() and XtVaSetValues(). You can query the value of Axis resources with XtGetValues() and XtVaGetValues(). In addition, there are some special purpose functions for controlling Axis numbering; these functions will be described later in this chapter.

Table 6-1. Axis Resources

Name	Class	Type	Default	CSG	Pg
Title					
XmNtitleFontList	XmCFontList	XmFontList	XmNfontList[a]	CSG	82
XmNtitlePSFontList	XmCPSFontList	XiPSFontList	XmNpsFontList[b]	CSG	82
XmNtitleString	XmCTitleString	XmString	NULL[c]	CSG	82
XmNtitle	XmCTitle	String	NULL	CSG	83
XmNtitleSize	XmCFontSize	Dimension	14	CSG	83
XmNtitleStyle	XmCFontStyle	String	NULL	CSG	84
XmNtitleColor	XmCForeground	Pixel	XmNforeground[d]	CSG	84
Labels					
XmNdrawLabels	XmCDrawLabels	Boolean	TRUE	CSG	84
XmNlabelFontList	XmCFontList	XmFontList	XmNfontList[a]	CSG	84
XmNlabelPSFontList	XmCPSFontList	XiPSFontList	XmNpsFontList[b]	CSG	84
XmNlabelSize	XmCFontSize	Dimension	10	CSG	85
XmNlabelStyle	XmCFontStyle	String	NULL	CSG	85
XmNlabelColor	XmCForeground	Pixel	XmNforeground[d]	CSG	85
Axis and Tick Marks					
XmNlineColor	XmCForeground	Pixel	XmNforeground[d]	CSG	85
XmNlineWidth	XmCLineWidth	Dimension	0	CSG	85
XmNmarkWidth	XmCLineWidth	Dimension	0	CSG	86
XmNmarksInside	XmCMarksInside	Boolean	FALSE	CSG	86
XmNmarksOutside	XmCMarksOutside	Boolean	TRUE	CSG	86
XmNmarkLength	XmCMarkLength	Dimension	5	CSG	86
XmNsubmarkLength	XmCMarkLength	Dimension	2	CSG	86
Frame and Origin					
XmNdrawFrame	XmCDrawFrame	Boolean	TRUE	CSG	87
XmNdrawOrigin	XmCDrawOrigin	Boolean	TRUE	CSG	87
XmNoriginWidth	XmCLineWidth	Dimension	0	CSG	87
XmNoriginColor	XmCForeground	Pixel	XmNforeground[d]	CSG	87
Grid					
XmNdrawGrid	XmCDrawGrid	Boolean	TRUE	CSG	88
XmNgridColor	XmCForeground	Pixel	XmNforeground[d]	CSG	88
XmNgridWidth	XmCLineWidth	Dimension	0	CSG	88
XmNgridPattern	XmCGridPattern	LinePattern	"\001"	CSG	88
XmNdrawSubgrid	XmCDrawSubgrid	Boolean	FALSE	CSG	89
XmNsubgridColor	XmCForeground	Pixel	XmNforeground[d]	CSG	89
XmNsubgridWidth	XmCLineWidth	Dimension	0	CSG	89
XmNsubgridPattern	XmCGridPattern	LinePattern	"\003"	CSG	89
Layout					
XmNmargin	XmCMargin	Dimension	2	CSG	89
XmNaxisWidth	XmCAxisWidth	Dimension	0	CSG	89
Auto Scaling[e]					

XmNautoScaleMin	XmCAutoScale	Boolean	TRUE	CSG	91
XmNautoScaleMax	XmCAutoScale	Boolean	TRUE	CSG	92
XmNautoScaleMargin	XmCAutoScaleMargin	Dimension	5	CSG	92
Bounds and Transform[e]					
XmNmax	XmCMax	Double	10.00	CSG	93
XmNmin	XmCMin	Double	0.00	CSG	93
XmNunitMultiplier	XmCUnitMultiplier	Double	1.00	CSG	94
XmNunitConstant	XmCUnitConstant	Double	o.0	CSG	94
XmNlog	XmCLog	Boolean	FALSE	CSG	96
Auto Marking[e]					
XmNautoMark	XmCAutoMark	Boolean	TRUE	CSG	97
XmNroundEndpoints	XmCRoundEndpoints	Boolean	TRUE	CSG	97
XmNpixelsPerLabel	XmCPixelsPerLabel	Dimension	50	CSG	98
XmNpixelsPerMark	XmCPixelsPerMark	Dimension	10	CSG	98
Manual Axis Marking[e]					
XmNlabelInterval	XmCLabelInterval	Short	1	CSG	100
XmNlabelOffset	XmCLabelOffset	Short	0	CSG	100
XmNmarkValues	XmCMarkValues	DoubleList	NULL	CSG	100
XmNnumMarks	XmCNumMarks	Cardinal	0	CSG	100
XmNmarkMultiplier	XmCMarkMultiplier	Double	1.0	CSG	100
XmNmarkConstant	XmCMarkConstant	Double	0.0	CSG	100
XmNmarkValueType	XmCMarkValueType	PlotDataType	XiPlotDataDouble	CSG	101
XmNmarkRecordSize	XmCMarkRecordSize	Cardinal	sizeof(double)	CSG	101
XmNmarkRecordOffset	XmCMarkRecordOffset	Cardinal	0	CSG	101
Auto Labeling[e]					
XmNautoLabel	XmCAutoLabel	Boolean	TRUE	CSG	101
XmNlabelFormat	XmCLabelFormat	String	"%g"	CSG	101
XmNlabelFunction	XmCLabelFunction	AxisLabelProc	NULL	CSG	101
Manual Axis Labeling[e]					
XmNlabels	XmCLabels	XmStringTable	NULL	CSG	105
XmNnumLabels	XmCNumLabels	Dimension	0	CSG	105
XmNlabelRecordSize	XmCLabelRecordSize	Cardinal	sizeof(String)	CSG	105
XmNlabelRecordOffset	XmCLabelRecordOffset	Cardinal	0	CSG	105

[a] *Inherited from the XmNfontList resource*

[b] *Inherited from the XmNpsFontList resource*

[c] *When an XmString resource is unset, its value is created from the corresponding String resource using a font tag derived from the corresponding style and size resources.*

[d] *Inherited from the Plotter SmNforeground resource*

[e] *Do not set these resources when using a DBPak subclass of this widget.*

6.2 Axis Resources

The Axis widgets control a number of important components of the overall Plotter display. Axis resources control the Axis title (strings like "Time (ms)"), the Axis labels (the strings that label the tick marks), the tick marks themselves, lines that form the axis, the origin, and a frame around the plotting area, and lines perpendicular to the axis that form a grid. Table 6-1 summarizes all the resources of the Axis widget.

The subsections below document the Axis resources that control the visual appearance of the axes. Figure 6-1 illustrates some of these visual resources. The resources that control how the axes are numbered, labeled, and marked, are explained in Section 6.3, *Controlling Axis Numbering*.

6.2.1 The Title

The "title" of an Axis is the string that identifies what the Axis is measuring. You might use strings like "Velocity (m/s)" or "Units Sold". If a title is specified, it will be drawn centered below the X axis, or centered on the left of the Y axis. The title of the Y axis is automatically drawn rotated 90 degrees counterclockwise so that it runs parallel to the axis and is read going up the axis. The title of the secondary Y axis is rotated 270 degrees so that it is read going down the axis.

The following resources control the attributes of the Axis title.

XmNtitleFontList

A font or list of fonts to be used to display the title for the Axis. If no value is specified for this resource, its default value is inherited from the parent Plotter XmNfontList resource.

XmNtitlePSFontList

The list of PostScript fonts to be used to display the title of the Axis in hardcopy. If no value is specified for this resource, its default value is inherited from the parent Plotter XmNpsFontList resource. You need only specify this resource if XmNtitleFontList contains fonts outside of the "standard" font families--Adobe Times, Helvetica, Courier, New Century Schoolbook, and Symbol.

XmNtitleString

The XmString to be displayed as the title of the Axis. If no XmString is specified for this resource, then one will be created using the XmNtitle string, and a font list tag constructed from the XmNtitleStyle and XmNtitleSize resources, as described in Section 3.3*, Fonts and Displayed Text.* Setting this resource will override any value previously created from the XmNtitle string. If you query this resource, the Plotter returns its private, internal copy of the XmString, which you must not modify or free.

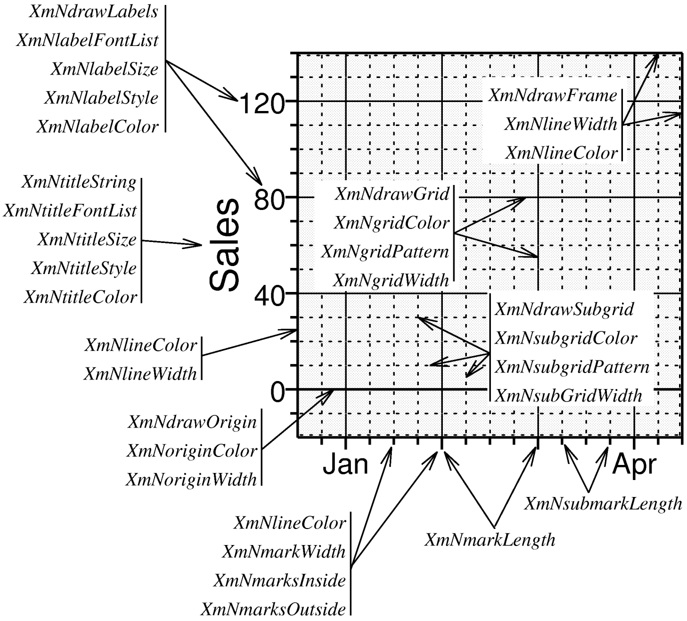


Figure 6-1. Resources that control Axis appearance

XmNtitle

This is the string to appear as the Axis title, if XmNtitleString is not also specified. The default is NULL, which specifies that no title is to be drawn. Note that this resource is an ordinary null-terminated character string, not a Motif XmString. When you set this resource, the string is converted to an XmString using a font list tag constructed from the XmNtitleStyle and XmNtitleSize resources, and this newly created XmString becomes the new value of the XmNtitleString resource. See Section 3.3 for details. When you set this resource, the Axis widget makes a private copy of the string, so the application may modify or free its copy of the string without affecting the widget. When you query this resource, you obtain this internal copy, which you must not modify or free.

XmNtitleSize

The point size of the Axis title. This value is used in the font list tag of the XmNtitle, if no XmNtitleString is specified. The default is 14.

XmNtitleStyle

The typeface for the Axis title. This resource is used in the font list tag of the XmNtitle, if no XmNtitleString is specified. XmNtitleStyle is a string resource, and the default value is NULL. If you set this resource, you will typically set it to strings such as "bold" and "italic", depending on the font tags in your XmNtitleFontList. When you set this resource, the string value is copied, and when you query it, you must not modify or free the returned value.

XmNtitleColor

The color to draw the Axis title in. The default value of this resource is inherited from the Plotter XmNforeground resource. Note, however, that changing the value of the XmNforeground resource after widget creation will not affect this resource.

6.2.2 The Labels

The "labels" of an Axis are the strings that are drawn beneath or to the left of the Axis tick marks. Usually these labels are numbers, but for some plots (a barchart of monthly sales, perhaps) they might be other strings, such as the names of months. Unlike the title, the labels of a Y axis are not rotated.

The following resources control the appearance of the Axis labels. (The content and position of the labels are controlled by other resources, described in Section 6.3, *Controlling Axis Numbering*).

XmNdrawLabels

A Boolean resource that specifies whether the labels should be drawn at all. The default is True. If set to False, then no labels will be drawn for the Axis, a feature that is occasionally useful.

XmNlabelFontList

A font or list of fonts to be used to display the labels for the Axis. If no value is specified for this resource, its default value is inherited from the parent Plotter XmNfontList resource.

XmNlabelPSFontList

The list of PostScript fonts to be used to display the labels of the Axis in hardcopy. If no value is specified for this resource, its default value is inherited from the parent Plotter XmNpsFontList resource. You need only specify this resource if XmNlabelFontList contains fonts outside of the "standard" font families--Adobe Times, Helvetica, Courier, New Century Schoolbook, and Symbol.

XmNlabelSize

The point size of the font to be used for the Axis labels. When labels are automatically computed by the Axis widget, or when labels are specified as strings (instead of XmStrings) with XiAxisSetLabels(), this resource is used along with XmNlabelStyle to construct a font list tag to specify which font in the XmNlabelFontList will be used to display the labels. The default is 10. See Section 3.3, *Fonts and Displayed Text* for more information.

XmNlabelStyle

The typeface to be used for the Axis labels. When labels are automatically computed by the Axis widget, or when labels are specified as strings (instead of XmStrings) with XiAxisSetLabels(), this resource is used along with XmNlabelSize to construct a font list tag to specify which font in the XmNlabelFontList will be used to display the labels. The default value is NULL. See Section 3.3, *Fonts and DisplayedText* for details. When you set this resource, the string value is copied, and when you query it, you must not modify or free the returned value.

XmNlabelColor

The color of the Axis labels. The default value of this resource is inherited from the Plotter XmNforeground resource. Note, however, that changing the value of the XmNforeground resource after widget creation will not affect this resource.

6.2.3 The Axis and Tick Marks

The basic feature of any Axis is a long horizontal or vertical line with tick marks drawn perpendicular to it. The Axis draws a long tick mark wherever it will also be drawing a label, and draws shorter lines for its unlabeled marks. In the Plotter widget lexicon, these line are referred to as "marks" and "submarks".

This subsection describes the resources that control the Axis line and its tick marks.

XmNlineColor

The color of the Axis line and the tick marks. (And also the color of the Axis frame, described below.) The default value of this resource is inherited from the Plotter XmNforeground resource. Note, however, that changing the value of the XmNforeground resource after widget creation will not affect this resource.

XmNlineWidth

The width, in pixels, of the Axis line. (And also the frame line, described below.) The default value is 0, which is a special value that produces efficiently-drawn lines one pixel wide. Using a line width of 2 or 3 produces bolder-looking axes and can add visual impact to graphs.

XmNmarkWidth

The width, in pixels, of the Axis tick marks. The default is 0, a special value that produces efficiently-drawn lines one-pixel wide. If you make the tick marks wider than this, you may also want to increase the width of the grid lines (which are described in the next subsection.) In particular, tick marks that are two-pixels wide can be troublesome, because there is no "center" pixel for a one-pixel wide grid to align with.

XmNmarksInside

A Boolean that specifies whether tick marks should be drawn on the "inside" of the Axis--above the X axis and to the right of the Y axis. The default is False. Setting this resource to True will cause tick marks to be drawn into the plotting area of the Plotter, where they may be obscured by actual plot data. Note that XmNmarksInside and XmNmarksOutside are independent resources; both may be True to produce tick marks that are drawn through the Axis, or both may be False to turn off the tick marks entirely.

XmNmarksOutside

A Boolean that specifies whether tick marks should be drawn on the "outside" of the Axis--below the X axis and to the left of the Y axis. The default is True, which is suitable for most graphs.

XmNmarkLength

The length, in pixels, of any tick marks that have labels. The default is 5. If both XmNmarksInside and XmNmarksOutside are True, then this many pixels will be drawn on both sides of the axis. Tick marks are drawn from the center of the axis outward; if you specify an XmNlineWidth more than one pixel wide, then a portion of the tick marks will be obscured by the axis line.

XmNsubmarkLength

The length, in pixels, of any tick marks that do not have labels. The default is 2. If both XmNmarksInside and XmNmarksOutside are True, then this many pixels will be drawn on both sides of the axis. Tick marks are drawn from the center of the axis outward; if you specify an Axis more than one pixel wide, then a portion of the tick marks will be obscured by the axis.

6.2.4 The Frame and the Origin

The Axis widget can optionally draw two other important lines, besides the main line of the axis itself. The first of these is the "frame", a line drawn parallel to the axis, on the opposite side of the plotting area. This line is drawn with the same color and width as the Axis itself, and if both X and Y Axis widgets of a plotter draw these "frame" lines, then the two axis lines plus the two frame lines form a complete box (or "frame") around the plotting area.

If the domain of the axis include zero, the X and Y Axis widgets will also optionally draw a line perpendicular to themselves to mark the zero point as the origin. Displaying an origin can be useful in some graphs, for example when the behavior of a graphed function differs in different quadrants of the real-number plane.

Note that the secondary Y axis, if it is displayed, never draws a frame or origin for itself.

XmNdrawFrame

A **Boolean** resource that specifies whether the Axis should draw a frame parallel to itself on the opposite side of the plotting area. The default is True. The frame is drawn using the same color and line width as the axis itself; these are specified by the XmNlineColor and XmNlineWidth resources described above.

XmNdrawOrigin

A Boolean that specifies whether the Axis should draw a perpendicular line across the plotting area to mark the points 0.0. The default is True, but the line will obviously only be drawn in the point 0.0 is between the axis minimum and the axis maximum. If the Axis will also be drawing a grid, you may want to set the origin width and/or color to ensure that the origin line is distinguishable from the grid lines.

XmNoriginWidth

The width, in pixels, of the origin line. The default is 0, a special value that specifies efficiently drawn lines one-pixel wide.

XmNoriginColor

The color of the origin line. The default value of this resource is inherited from the Plotter XmNforeground resource. Note, however, that changing the value of the XmNforeground resource after widget creation will not affect this resource.

6.2.5 The Grid

The X and Y Axis widgets will optionally draw lines perpendicular to the axis across the plotting area of the Plotter. These lines are aligned with the tick marks on the axis. If both X and Y Axis widgets draw these lines, they form a grid, and give the plotting area the appearance of graph paper. Each Axis draws two sets of grid lines, which can have different colors, thicknesses, and dash patterns. One set of lines is drawn for every labeled mark. These lines are known as the "grid", and are relatively widely spaced. The other set of lines is drawn for the remaining unlabeled marks (the "submarks"). These lines form the "subgrid". They are closely spaced, and if they are drawn at all, they are generally drawn using a fainter color, or with a widely spaced dot pattern.

Note that the secondary Y axis, if displayed, never draws a grid or subgrid for itself, and so the resources described in this section have no effect for the secondary Y axis.

This subsection describes the Axis resources that control the grid and subgrid.

XmNdrawGrid

A Boolean that specifies whether the grid lines should be drawn. The default is True.

XmNgridWidth

The width, in pixels, of the grid lines. The default is 0, a special value that specifies efficiently-drawn lines, one pixel wide.

XmNgridColor

The color of the grid lines. The default value of this resource is inherited from the Plotter XmNforeground resource. Note, however, that changing the value of the XmNforeground resource after widget creation will not affect this resource. A good choice for this resource is a color that is distinguishable, but not too distinct in hue or intensity, from the XmNplotAreaColor of the Plotter widget. This will result in a grid that is visible, but does not distract from the actual plot data. On monochrome systems, you can use the XmNgridPattern resource to simulate this effect.

XmNgridPattern

The line pattern to use for the grid lines. The default is the "\001\001", a one pixel on, one pixel off pattern. The value NULL specifies solid lines. Like all String resources, any pattern you specify for this resource is copied by the Axis widget, so you may modify or free the pattern once you have set it. If you query the resource, however, you must not modify or free the internal copy of the string that is returned. If you are using a color display, you may prefer to use the XmNgridColor resource and set this resource to NULL.

XmNdrawSubgrid

A Boolean resource that specifies whether the subgrid lines should be drawn. The default is False. If this resource is set to True, and XmNdrawGrid is also True, then the subgrid lines will be drawn for every unlabeled tick mark on the axis. If this resource is True, and XmNdrawGrid is False, then the subgrid lines will be drawn for both labeled and unlabeled tick marks.

XmNsubgridWidth

The width in pixels of the subgrid lines. The default is 0.

XmNsubgridColor

The color of the subgrid lines. The default value of this resource is inherited from the Plotter XmNforeground resource. See XmNgridColor above for more information.

XmNsubgridPattern

The line pattern for the subgrid lines. The default is "\001\003"--one pixel followed by three pixels off. See XmNgridPattern above for more information about line pattern resources.

6.2.6 Axis Layout

There are two Axis resources that allow fine control over the positioning of the labels and title in the Axis, and the positioning of the Axis within the Plotter. They are explained below:

XmNmargin

This resource specifies the distance, in pixels, between the axis tick marks and their labels, the distance between the tallest or widest of the tick labels and the axis title, and the margin between the axis title and the edge of the plotter. (At the edges, the Axis XmNmargin resource is added to the Plotter XmNmarginWidth or XmNmarginHeight resource.) The default is 2 pixels. Larger values give the axis a more open appearance, and are appropriate when using larger font sizes.

XmNaxisWidth

This resource serves a purpose similar to the Plotter XmNtitleMinHeight and XmNlegendMinWidth resources. "Axis width" in this context means the short dimension of the axis--the height of an X axis, or the width of a Y axis. The default value of 0 specifies that the Axis should compute its own width, and that the Plotter should allocate this much screen space for displaying the Axis. If you specify a non-zero value for this resource, the plotter will allocate that many pixels for the axis. This resource differs from the Plotter XmNtitleMinHeight and

XmNlegendMinWidth in that the Axis will not ignore it if it is too small. If you specify a value for XmNaxisWidth that is too small, portions of the Axis will be chopped off by the edges of the Plotter window.

The purpose of this resource is to allow you to lineup the axes of adjacent plotter widgets, even if those widgets use different fonts or have labels of different lengths. By specifying the same value for the Axes of two adjacent Plotter widgets, you can guarantee that the axes will appear at the same distance from the bottom or left edges of the Plotter. If you use this technique, note that the Y axis can be troublesome: since the labels are drawn horizontally, the width of the axis depends on the content of the individual labels, which generally depends on the axis bounds. If your application allows the Y axis bounds to vary over several orders of magnitude you may be better off letting the Axis specify its own width.

6.3 Controlling Axis Numbering

The resources described in the section above control such things as the length and width of the Axis tick marks and the size of the font used for the Axis labels. This section explains how you can specify the *position* of the tick marks and the *content* of the labels.

For many applications, the X and Y Axis widgets of a Plotter will perform automatically, and you will never need to manipulate them at all. By default, the Axis will set its minimum and maximum bounds to be large enough to accommodate all the data currently displayed in the Plotter. Once the Axis has performed this "autoscaling", it will do "auto-marking"--it will place a reasonable number of small and large tick marks at intervals along the axis. In this default automatic setup, the Axis will automatically label each of the large tick marks with the value it measures. The tick marks are chosen so that these will always be "round" numbers--multiples of 1, 2, 5, 10, 20, 50, etc.

The Axis widget works well in this fully automatic mode for many applications, but there are times when you may want to turn off auto-scaling (when you want to "zoom-in" on some data, for example) or turn off auto-marking (when autoscaling is off, and the user has requested a particular mark interval, perhaps), or turn off auto-labeling (usually when you want to label an axis with the names of months or other strings rather than with numbers). It doesn't make sense to use auto-scaling without auto-marking and auto-labeling, or to use auto-marking without auto-labeling, so there are four modes that you can use the Axis widget in (and four functions for configuring the Axis in those modes).

- **Autoscale Mode**. This is the fully automatic mode. The Axis chooses the bounds, computes mark positions, and labels the marks. The function XiAxisAutoscale() places an Axis in this mode.
- **Automark Mode**. In this mode you specify the minimum and maximum values for the Axis, but it automatically computes reasonable mark positions and labels

the marks. The function XiAxisSetBounds() takes a minimum and maximum values as arguments, and places the Axis in this mode.

- **Autolabel Mode**. In this mode, you specify the axis bounds, and the positions of the marks. The Axis automatically provides numeric labels for each mark. The function XiAxisSetMarks() places an Axis in this mode.
- **Manual Mode**. In this mode you specify the bounds and mark positions, and also provide an array of strings to use as labels for each mark. The function XiAxisSetLabels() places an Axis in this mode.

In addition, the Axis widget supports a somewhat different "log mode" which automatically marks and labels the axis with a logarithmic transform. The subsections below document the convenience functions and Axis resources you can use to configure an Axis in each of the modes.

6.3.1 Computing Axis Bounds Automatically

By default, an Axis will set its bounds to be large enough for whatever data is displayed in the Plotter. When auto-scaling is on, auto-marking and auto-labeling must also be on, because the axis must be able to compute new mark positions and labels as the bounds of the axis change. The resources that control auto-marking and auto-labeling are described in the following subsections; you can use any of them when you are using auto-scaling.

The easiest, and the recommended, way to place an Axis into this fully automatic mode (assuming you left this default mode for some reason) is to call the function **XiAxisAutoscale().**

```
void XiAxisAutoscale(Widget w);
```

`w` An Axis widget that should have high and low bounds autoscaled.

Calling this function turns on autoscaling for both the minimum and maximum bounds of the axis, and turns on auto-marking and auto-labeling as well. It also sets the XmNroundEndpoints resource to True; this resource is described below.

The following resources specify the auto-scaling behavior of an Axis; you can set them individually to give you finer control over auto-scaling than is possible with XiAxisAutoscale().

XmNautoScaleMin

A Boolean that specifies whether the minimum bound of the axis should be automatically adjusted to fit all the data in the Plotter widget. The default is True. Most often, you will set XmNautoScaleMin and XmNautoScaleMax to the same value, but there are times when you will want to set them independently: when displaying a bar chart, for example, you generally want the minimum value on the Y axis to be zero, regardless of the data values.

XmNautoScaleMax

A Boolean that specifies whether the maximum bound of the axis should be automatically adjusted to fit all the data in the Plotter widget. The default is True. See also XmNautoScaleMin.

XmNautoScaleMargin

When the Axis computes it bounds by autoscaling, it finds the largest and smallest data values (for the X or the Y dimension) and adds a margin to these values. This resource is a positive integer that specifies the size of this margin as a percentage of the total range of the axis. The default value is 5. With data that ranged between 100 and 300, and this default margin of 5, for example, the minimum bound of an Axis would be auto-scaled (if XmNautoScaleMin were True) to 90, and the maximum bound would be scaled to 310 (because 5% of 300-100 is 10).

The minimum and maximum bounds of an Axis may also be automatically adjusted depending on the XmNroundEndpoints resource. This resource is used as part of the auto-marking process and is described below.

6.3.2 Setting Axis Bounds Manually

There are times when axes should not be autoscaled. If you implement a "zoom-in" feature in your application, for example, you will probably have to turn off auto-scaling for this. Or if you have a lot of data stored in a single plot, you might only display a portion of it at a time, and allow the user to pan through the data with a scrollbar or similar technique.

```
void XiAxisSetBounds(Widget w, double min, double max);
```

`w`	An XiAxis widget.
`min`	The new minimum bound of the widget.
`max`	The new maximum bound of the widget.

You can set the bounds on a Axis by calling XiAxisSetBounds(). This function turns off autoscaling of both the minimum and maximum bounds, and sets those bounds to the specified values. It also sets the XmNroundEndpoints resource (described below) to False, and turns auto-marking and auto-labeling on. (If you do not want auto-marking or auto-labeling, there are other functions, described below, that let you set the bounds and the mark positions, or the bounds, mark positions, and labels.)

```
void XiAxisGetBounds(Widget w, double *min, double *max);
```

`w`	An XiAxis widget.
`min`	Returns the minimum bound of the widget.
`max`	Returns the maximum bound of the widget.

Whether or not the Axis bounds are auto-scaled, you can query their values with XiAxisGetBounds(): This function returns the minimum and maximum bounds of the specified Axis at the specified addresses.

XiAxisSetBounds() and XiAxisGetBounds() are the recommended (and usually the easiest) techniques forsetting and querying axis bounds. You can, however, set these values individually with the following resources:

XmNmin

Specifies the minimum value for the Axis. Note that this resource is a double, so special care is required when setting it.

XmNmax

Specifies the maximum value for the Axis. Note that this resource is a double, so special care is required when setting it.

Note that the minimum and maximum bounds that you specify for an axis may be adjusted during auto-marking, depending on the XmNroundEndpoints resource. This resource is described below.

6.3.3 Setting Secondary Y Axis Bounds

The bounds of the secondary Y axis (if one is displayed) are always based on the bounds of the primary Y axis: the maximum value of the secondary axis is linearly proportional to the maximum value of the primary axis, and the minimum values of the axes are similarly proportional. I.e. for any possible bounds of the primary and secondary Y axes, we can always find a multiplier m and a constant c such that:

$$\text{y2 max} = m\ x\ ymax + c$$

$$\text{y2 min} = m\ x\ ymin + c$$

In order to use a secondary Y axis, the multiplier *m* must be specified as the XmNunitMultiplier resource of the secondary Y axis, and the constant *c* must be specified as the XmNunitConstant resource of the secondary Y axis:

XmNunitMultiplier

A double that specifies one half of the linear equation that defines the relationship between values on the primary Y axis and values on the secondary Y axis.

XmNunitConstant

A double that specifies the other half of the linear equation that defines the relationship between values on the primary Y axis and values on the secondary Y axis.

When these two resources are specified for the secondary Y axis, the bounds for that axis will automatically be updated according to the equations above every time the bounds of the primary Y axis change. Note that the default multiplier is 1.0 and the default constant is 0.0; this means that by default, the secondary Y axis will display the same values and the primary Y axis.

Setting these resources is simple enough. Determining the appropriate multiplier and constant can be confusing, however. When you know the desired minimum and maximum bounds for both axes, you can compute *m* and *c* as follows:

$$m = \frac{\{y2\ \mathrm{max} - y2\ \mathrm{min}\}}{\{y\ \mathrm{max} - y\ \mathrm{min}\}}$$

$$c = y2\ \ \mathrm{min} - m\ \mathrm{x}\ y\ \mathrm{min}$$

The most common use for a secondary Y axis is to display a second set of units for the data. For example, when graphing temperatures, you might display degrees Celsius on the primary Y axis and degrees Fahrenheit on the secondary Y axis. Or, when graphing energy values, you might display kilojoules on one Y axis and kilocalories on the other, or you might express volume in liters and gallons, or monetary value in German marks and Japanese yen. Instead of simply setting a separate set of minimum and maximum bounds for the two axes, however, you specify a linear transformation that relates the two units and converts the bounds of the primary axis into the bounds of the secondary axis. To return to our temperature example, you could specify an XmNunitMultiplier of 1.8, and an XmNunitConstant of 32.0 to convert degrees Celsius on the primary Y axis to degrees Fahrenheit on the secondary Y axis. The benefit of specifying the secondary Y axis bounds this way is that they will automatically be updated appropriately when the bounds of the primary Y axis change, when autoscaling occurs, for example.

The important thing to remember whenever you use a secondary Y axis is that all the data you graph must be expressed in the units used for the primary Y axis. So, if you have degrees Celsius on the primary Y axis, then all your temperature data should be expressed in those units. If you do happen to have some data in degrees Fahrenheit, however, you don't actually have to convert it all to Celsius yourself. Recall that the XiPlotData structure (introduced in Chapter 4, *Plot Data Representation*) which is used to store data for plotting allows a linear transformation to be applied to any data. Thus, you'd need simply to figure out an appropriate multiplier and constant to convert degrees Fahrenheit to degrees Celsius. If you were using a LinePlot widget, you'd specify an XmNyValueMultiplier of 0.55555 and an XmNyValueConstant of -17.777 to do this. (These figures were computed simply by reversing the y and y2 terms in the above equations.)

There is another way you might use a secondary Y axis. Sometimes, you may want to graph two unrelated quantities in the same Plotter widget, simply for the convenience of having them displayed in the same place. For example, in an application that monitors the flow of liquid going through a pipe, you might want to graph both temperature and pressure on the same graph. While this kind of graph can be confusing to read, many disciplines have standardized data display formats that do this sort of thing.

The difficulty of displaying both temperature and pressure on the same graph is that they are unrelated units, and there is no natural linear transformation that converts one to the other. We can still produce a graph like this, however; it will just take some work. Suppose we are displaying pressure in psi (pounds per square inch) on the primary Y axis, and temperature in degrees Celsius on the secondary Y axis. Further, suppose that we want our Y axis to display pressures between 0 psi and 100 psi, and that we want the secondary Y axis to display temperatures between 20 C and 70 C. To do this, we would set the bounds of the primary Y axis directly, and set the XmNunitMultiplier of the secondary Y axis to 0.5 and the XmNunitConstant of that axis to 20.0.

With the axes set up like this, we can go ahead and graph our pressure data as normal, since pressure is displayed on the primary Y axis. But remember that all graphs are displayed in terms of the units on the primary Y axis. So to graph temperature we first have to convert our temperature values to pressure values. Obviously, degrees Celsius cannot be converted to pounds per square inch. In our limited case, however, all we really need to do is scale a number between 20 and 70 (our temperature) to a proportional number between (0 and 100). We do this simply by multiplying by 2 and adding -40. As we saw above, we can automatically scale LinePlot data in this way by setting the LinePlot XmNyValueMultiplier to 2.0 and setting the LinePlot XmNyValueConstant to -40.0.

So, with a linear transform applied to our secondary Y axis which displays temperature, and with the reverse linear transform applied to all of our temperature data, we can successfully graph pressure and temperature data in the same Plotter widget. Because the primary and secondary axes are linked, if either pressure or temperature data causes the Plotter to autoscale, both axes will be updated. Or, if you change the bounds on the primary (pressure) axis, then the bounds on the secondary (temperature) axis will be updated as well. In this case, when we are graphing distinct quantities, we are more likely to want to change the bounds of one axis without changing the bounds of the other. This is where this case gets more complicated. To change the bounds of the pressure axis without changing the bounds on the temperature axis, we need to recompute the linear transform for the temperature axis (the secondary axis) and also recompute the linear transforms for all the plots that display temperatures.

Finally, note one last thing about the secondary Y axis. The secondary axis is assumed to be of secondary importance to the primary axis, and thus it never draws grid lines, or an origin or frame line, as the primary axis does.

6.3.4 Specifying Logarithmic Axes

The Axis widget can display itself logarithmically scaled, which makes it easy to produce semi-log and log-log plots, as shown in Figure 6-2. The Axis widgets control the scaling of data points to appear on the screen, so when an Axis is in log mode, any data that appears in the Plotter will automatically be scaled correctly against the axis.

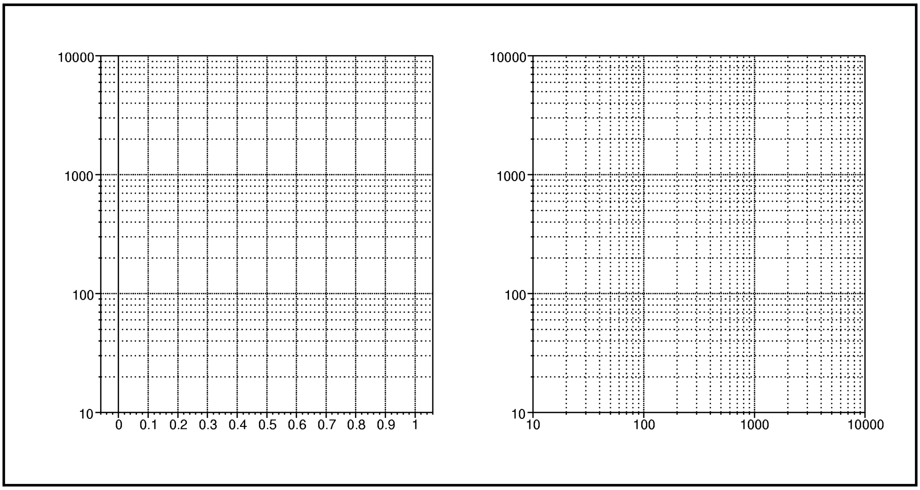


Figure 6-2. Plotter widgets with logarithmic axes

The XmNlog resource controls this feature:

XmNlog

A Boolean resource that specifies whether the axis will display itself logarithmically (base 10) or not. If True, then the Axis will automatically divide its range into decades, and will place its marks and grid lines appropriately. The Axis widget controls the scaling of Plot data, so plots will automatically be displayed correctly on the logarithmic axes.

Note that logarithmic axes can only be used when auto-labeling and auto-marking are on. Also, since the logarithm of zero and all negative numbers is undefined, you cannot have a logarithmic axis with a non-positive minimum bound, and you should not attempt to plot non-positive data against a logarithmic axis.

There is no special function for making an Axis logarithmic; you can do so simply by setting the XmNlog resource. It is recommended that you do this in conjunction with XiAxisSetBounds(), which ensures that auto-labeling and auto-marking are turned on, and that autoscaling is turned off (you can use autoscaling with logarithmic axes, but you may get better looking results if the axis bounds are always an exact power of 10.)

The Y axis of the semi-log plot shown in Figure 6-2, for example, was set up with the following code:

```
yaxis = XiPlotterYAxis(plotter);
XiAxisSetBounds(yaxis, 10.0, 10000.0);
XtVaSetValues(yaxis,
              XmNlog, True,
              XmNdrawSubgrid, True,
              NULL);
```

Note that logarithmic axes have labeled ticks every decade, and have unlabeled ticks between decades. The labels are evenly spaced, but the unlabeled ticks are not. For this reason, you may want to turn on the subgrid with the XmNdrawSubgrid resource to get the effect of logarithmic graph paper.

6.3.5 Computing Axis Marks Automatically

By default, the Axis widget will automatically compute positions for a reasonable number of labeled and unlabeled tick marks. Calling either of the functions XiAxisAutoscale() and XiAxisSetBounds() will also place an Axis into the auto-marking mode. Note that when auto-marking is on, auto-labeling must also be on. This is because if an application does not know where the Axis will be placing the marks, it cannot know what labels to provide for those marks.

The following resources control auto-marking of the Axis widget.

XmNautoMark

A Boolean that specifies whether an Axis will have its mark positions automatically computed. The default is True. You can set this resource directly, but the recommended way for specifying automatic marking is to call XiAxisAutoscale() or XiAxisSetBounds(). The recommended way to turn automatic marking off is by calling XiAxisSetMarks() or XiAxisSetLabels(). (documented below.)

XmNroundEndpoints

A Boolean that specifies whether the Axis may adjust its minimum and maximum bounds in the process of computing marks so that they are multiples of the mark interval and a mark (either labeled or unlabeled) will appear at each bound.

The default is True. This resource is set to True by calling XiAxisAutoscale(), and is set to False by calling XiAxisSetBounds(), XiAxisSetMarks(), or XiAxisSetLabels(). If the bounds are adjusted, they are always adjusted "outward"--the minimum bound is made smaller, and the maximum bound made larger.

XmNpixelsPerLabel

This resource is an integer that provides a hint to the Axis auto-marking code about how far apart the longer, labeled tick marks should be placed. This resource can only be a hint because the Axis always places its marks at round numbers. The default is 50 pixels. Increasing this number will generally result in labeled marks that are further apart, and decreasing it will result in labeled marks that are closer together.

XmNpixelsPerMark

This resource is an integer that provides a hint to the Axis auto-marking code about how far apart the shorter, unlabeled tick marks (the "subticks") should be placed. The default is 10 pixels. This resource can only be a hint because the Axis divides the interval between two labeled ticks into 2, 4, 5, or 10 sub-intervals.

6.3.6 Specifying Axis Marks Explicitly

When you specify fixed bounds for an axis, and don't want the uncertainty of not knowing what space between tick marks the Axis would choose automatically, you can specify the interval between the tick marks, and the interval between labeled tick marks explicitly. The easiest, and the recommended way to do this is with the function XiAxisSetMarks(), shown in Figure 6-3.

```
void XiAxisSetMarks(Widget w, double min, double max,
                     double mark_interval, double first_mark,
                     int label_interval, label_offset);
```

`w`   The Axis widget.

`min, max`

The minimum and maximum bounds for the Axis.

`mark_interval`

The interval (in axis units) between successive tick marks, whether labeled or unlabeled.

`first_mark`

The position of the first mark on the Axis. Subsequent marks will be offset from this first one by a multiple of the mark_interval argument.

`label_interval`

The interval between labeled marks. Unlike the mark_interval argument, label_interval is an integer and is measured in multiples of the mark interval itself. This argument must be greater than or equal to 1.

`label_offset`

The number of marks to skip before placing the first label. This argument must always be less than label_interval.

Figure 6-3. XiAxisSetMarks()

XiAxisSetMarks() sets the bounds and marks you specify for the axis. It turns off autoscaling on both the minimum and maximum bounds of the axis, turns off automarking, and sets XmNroundEndpoints to False. It also turns on auto-labeling, so the marks you specified to be labeled will automatically receive their correct numeric labels.

If you wanted, for example, to set up an axis running from -3.5 to 3.5, with marks every .25 units, and labels on the whole numbers, you might use XiAxisSetMarks() as follows:

```
XiAxisSetMarks(XiPlotterXAxis(plotter), -3.5, 3.5, 0.25, -3.5, 4,
2);
```

The last two arguments, label_interval and label_offset require some extra clarification: these arguments are integers, and rather than being measured in the coordinate system of the axis, they are interpreted as multiples of the mark_interval argument. Thus we set label_interval to 4 to specify that every fourth mark should have a label. We set label_offset to 2 in the example above to specify that the first two marks on the axis will be unlabeled. This means that the marks at -3.5 and -3.25 will not be labeled, but the mark at -3.0 will be. Subsequently, every fourth mark will be labeled, and since the mark_interval is 0.25, and 4\(mu0.25 is 1.0, the other labels will appear at -2.0, -1.0, 0.0, and so on.

The positioning of marks can also be controlled with Axis resources. Note that you can use these resources to specify an arbitrary array of mark positions if, for some reason, you do not want an Axis to be uniformly marked. The resources for specifying this array of positions are the fields of an XiPlotData structure--this is the same way you specify arrays of values for plots. The resources allow you to use values of just about any type, stored at arbitrary offsets in an array of structures. See Chapter 4, *Plot Data Representation* for more information on specifying this kind of array of values.

XmNlabelInterval

The interval between labeled marks, measured in multiples of the mark interval itself. This resource is the equivalent of the label_interval argument of XiAxisSetMarks() described above.

XmNlabelOffset

The number of marks to skip before labeling the first one. The equivalent of the label_offset argument described above.

XmNnumMarks

The number of marks to appear on the Axis. If you use XiAxisSetMarks(), this value is computed for you from the axis bounds, the position of the first mark, and the mark interval. When specifying marks directly with resources, you must compute it yourself and set it on this resource.

XmNmarkMultiplier

The interval, in axis units, between marks on the axis. This is equivalent to the mark_interval argument described above.

XmNmarkConstant

The position of the first mark. This is the equivalent to the first_mark argument described above.

XmNmarkValues

A pointer to the first element in an array of structures that contain an explicit list of mark positions. The following resources specify the type of these values and how to extract them from the structures. You only need to use this and its related resources if you want marks on an Axis that are not linearly spaced. If you specify this explicit array of values, XmNmarkMultiplier and XmNmarkConstant will be used to perform a linear transform on the values; set those resources to 1.0 and 0.0 respectively (their default values) if you do not want the mark position values changed.

XmNmarkValueType

An XiPlotDataType that specifies the type of the data in the XmNmarkValues array.

XmNmarkRecordSize

The size, in bytes, of each of the structures in the XmNmarkValues array. Use sizeof() to compute this size. Note that it is perfectly legal to use an array of doubles or integers, or other types that are not actually structures.

XmNmarkRecordOffset

The offset of the mark position within each element of the XmNmarkValues array. Use XtOffsetOf() to compute this value. If XmNmarkValues is an array of doubles or integers, or some other simple type, then this resource will be 0.

6.3.7 Computing Axis Labels Automatically

All the axis modes and functions we've discussed so far have relied upon automatic Axis labeling. Unlike the problem of choosing a reasonable spacing for axis marks, the problem of placing numeric labels on marks is straightforward, and there is never a reason to turn off auto-labeling unless you will be using non-numeric labels, such as the names of months.

The resources you can use to control the labels that the Axis automatically generates are described below.

XmNautoLabel

A Boolean that specifies whether the Axis automatically provides labels for its marks. The default is True. You can set this resource directly, but usually you will rely on the Axis setup functions to do it for you. XiAxisAutoscale(), XiAxisSetBounds(), and XiAxisSetMarks() all set this resource to True. XiAxisSetLabels() sets it to False.

XmNlabelFormat

A printf() format string used by the Axis to convert the numeric value of a mark position to the equivalent string. The default is "%g". Like all String resources, the Axis makes a copy of any string you specify, so you may modify or free the string after setting this resource. If you query the resource, you must not modify or free the private value you obtain.

XmNlabelFunction

A function to call, instead of sprintf() to convert the numeric position of a tick mark to the label for that mark. The default value for this resource is NULL, which means that the Axis will use sprintf() with the format string specified by

XmNlabelFormat. If you specify this resource, then sprintf() will never be called, and XmNlabelFormat will be ignored. The type of this resource is XiAxisLabelProc:

```
typedef String (*XiAxisLabelProc)(XiAxisWidget, double);
```

A procedure of this type expects an XiAxisWidget as its first argument and a double mark position as its second. It should return a string, in allocated memory (use XtMalloc() or XtNewString()), that will serve as the label for the mark at the specified position. Once this label function has allocated memory for the label and returned it, it should never again modify or free that memory; the Axis widget will automatically free it (with XtFree()) when appropriate.

One common use for this resource is to multiply or divide the mark position by some power of ten. If an X axis is labeled "Time (ms)", for example, but the plot data is actually specified in seconds, rather than milliseconds, then you could multiply the label positions by 1000 to convert. Or if a Y axis is labeled "Profit (millions)", you might divide by 1,000,000. Doing this would result in more compact labels, and remove redundant information (repeated zeros on each label) from the Plotter display.

There are other useful, if less simple, conversions you can do with the XmNlabelFunction. If you have data measured in seconds, and a labeled tick mark every 900 seconds, for example, you might use this resource to supply a function that would convert these times to clock format and produce labels like "10:30am" and "4:45pm".

6.3.8 Specifying Axis Labels Explicitly

In some circumstances, notably with bar plots, you need an axis that has textual labels rather than numeric ones. The X axis of a bar plot might be labeled, for example, with the names of months, or the quarters of a fiscal year, or the hostnames of computers on a local area network. In order to label an Axis in this way, you must supply an array of strings for it to use as labels. Note that this cannot work when auto-scaling or auto-marking is in effect for the Axis.

Even when you are not using numeric labels, the Axis still operates with a minimum and maximum bound and a mark interval. Generally, the easiest way to handle this is to place your labels at integral values starting at 1.0. This is a good choice because these are the X values that the BarPlot widget, by default, assigns to its Y data. You do need to be careful, when using non-numeric axes with BarPlots or LinePlots without X values, that the labels and the data points line up.

It does not make sense to have unlabeled marks between marks that are labeled with discrete quantities such as computer names, so you will want a mark interval of 1.0 and a label interval of 1 in this case. If you were using the names of months as your labels, it might make sense to have unlabeled marks between them (perhaps only to have a subgrid showing in the Plotter) and you could specify this with a different combination of mark interval and label interval.

The easiest, and the recommended, way to specify labels for an Axis is with XiAxisSetLabels() or XiAxisSetLabelStrings() , shown in Figure 6-4.

The first seven arguments of both functions have the same names and meanings as the arguments to XiAxisSetMarks(), documented above. The remaining two arguments specify an array of labels, and the number of elements in that array. For XiAxisSetLabels(), the eighth, *labels* argument is an array of NULL-terminated strings. XiAxisSetLabels() converts this array of strings to an internal array of XmStrings using a font list tag constructed from the XmNlabelStyle and XmNlabelSize resources. Note that this is a one-time conversion--when labels are specified with XiAxisSetLabels(), changing the value of XmNlabelStyle or XmNlabelSize will not change the font used to display the labels. Once XiAxisSetLabels() has been called, the Axis widget makes no further use of the *labels* array, and you can modify or free it.

XiAxisSetLabelStrings() takes an array of XmStrings as its eighth argument, instead of an array of strings. These strings do not need to be converted, and are displayed directly by the Axis widget. Note that the Axis widget does not make a copy of these XmStrings or of the array that holds them. You must not modify or free the strings or the array until the Axis will no longer be using them (until you've specified a new array, or switched the axis back to auto-label mode).

For both functions, the ninth argument is the number of elements in the array of strings or XmStrings. This number should match the number of labeled marks on the axis, which is implicitly specified by the combination of the second through seventh arguments.

You could use XiAxisSetLabels() as follows, for example, to create an X axis labeled with the names of months:

```
static char *months[] = {"Jan", "Feb", "Mar", "Apr", "May"};
XiAxisSetLabels(XiPlotterXAxis(plotter),
                0.25, 5.75, 1.0, 1.0, 1, 0,
                months, 5);
```

The numeric positions of the marks and labels is arbitrary, since this axis isn't numbered. As recommended above, however, this example places the marks at an interval of 1.0(mark_interval) starting at 1.0 (first_mark). Each mark is labeled (label_interval = 1), starting with the first one (label_offset = 0). The bars of a BarPlot widget are, by default, positioned at an interval of 1.0, starting at 1.0, so this axis will work well with a BarPlot widget. Note that the axis bounds are not set to 1.0 and 5.0. This is because bars in a BarPlot have width--a bar positioned at 1.0 could extend between 0.5 and 1.5, and a bar positioned at 5.0 could extend between 4.5 and 5.5. Here the axis bounds are set to 0.25 and 5.75 to provide a margin on either side of these bars.

```
void XiAxisSetLabels(Widget w, double min, double max,
                      double mark_interval, double first_mark,
                      int label_interval, label_offset,
                      String *labels, Cardinal num_labels);
void XiAxisSetLabelStrings(Widget w, double min, double max,
                            double mark_interval, double first_mark,
                            int label_interval, label_offset,
                          XmString *label_strings, Cardinal num_labels);
```

`w` The XiAxis widget that is having its labels set.

`min, max`

The minimum and maximum bounds for the axis.

`mark_interval, first_mark`

The positions of each of the marks; see XiAxisSetMarks() above.

`label_interval, label_offset`

The positions of each of the labels; see XiAxisSetMarks() above.

`labels`

For XiAxisSetLabels() only, an array of String to use as labels for the axis. The strings will be converted to an internal array of XmString, using a font list tag derived from XmNlabelStyle and XmNlabelSize. Once converted, the widget never uses these strings or their array again, so you may modify or free them.

`label_strings`

For XiAxisSetLabelStrings() only, an array of XmString to use as labels for the axis. The Axis widget does not make a copy of the array, or of the XmStrings it contains, so you must not modify or free them until they are no longer in use (when you've specified a new array, or switched back to auto-label mode.)

`num_labels`

The number of labels for the axis. There must be at least this many strings in the labels or label_strings array, and this value must match the number of labels implicitly specified by the other arguments to this function.

Figure 6-4. XiAxisSetLabels() and XiAxisSetLabelStrings()

As with XiAxisSetMarks(), XiAxisSetLabels() and XiAxisSetLabelStrings() turn off autoscaling on the minimum and maximum bounds, turn off auto-marking, and set XmNroundEndpoints to False. Unlike XiAxisSetMarks(), they also turn off auto-labeling.

You can also specify a set of non-numeric labels through resources. The resources used to specify the positioning of marks and labeled marks were described above. The resources below let you specify an array of labels in a more flexible way (using an XiPlotData-style scheme) than is possible with XiAxisSetLabels(). Note that these resources will be ignored (and overwritten) if XmNautoLabel is True.)

XmNlabels

An array of XmStrings, each of which is a label, or an array of structures each of which contains an XmString label.

XmNnumLabels

The number of elements in the XmNlabels array.

XmNlabelRecordSize

The size, in bytes, of each element in the XmNlabels array. Use sizeof() to specify this value. If XmNlabels is a simple array of XmStrings, this resource will be sizeof(XmString). The Axis will use this value to move from element to element within the array.

XmNlabelRecordOffset

The offset, in bytes, within each element of the array, of the label. If XmNlabels is a simple array of XmStrings, this resource will be 0. Otherwise, use XtOffsetOf() to compute the offset of the label field within the structure. The Axis will use this offset to extract the label from the structure.

6.4 World <--> Pixel Coordinate Conversion

On occasion you may want to convert a point from floating-point world (or axis) coordinates to the equivalent integer pixel coordinates, or vice versa. The two public functions documented below allow you to do this quite easily.

```
void XiConvertWorldToPixel(Widget plotter, double x, double y,
                            int *px_return, int *py_return);
```

`plotter` The XiPlotter widget for which the conversion is to be done.

`x` The X axis coordinate of the point to be converted.

`y` The Y axis coordinate of the point to be converted.

`px_return`

The address at which the X pixel coordinate of the point is to be returned.

`py_return`

The address at which the Y pixel coordinate of the point is to be returned.

```
void XiConvertPixelToWorld(Widget plotter, int px, int py,
                            double *x_return, double *y_return);
```

`plotter` The XiPlotter widget for which the conversion is to be done.

`px` `The X pixel coordinate of the point to be converted.`

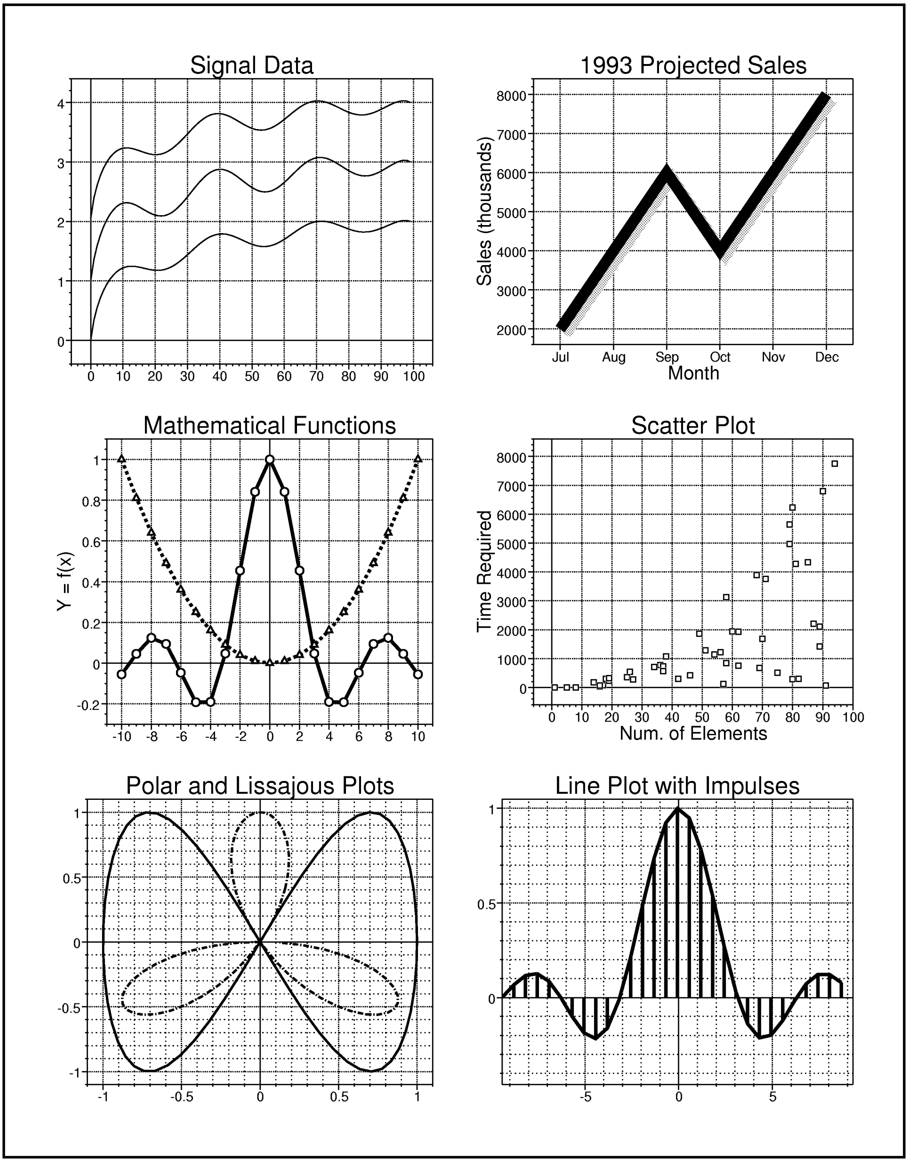
`py` `The Y pixel coordinate of the point to be converted.`

`x_return`

The address at which the X axis coordinate of the point is to be returned.

`y_return`

The address at which the Y axis coordinate of the point is to be returned.


Signal Data
0
1
2
3
4
0 10 20 30 40 50 60 70 80 90 100
1993 Projected Sales
Sales (thousands)
2000
3000
4000
5000
6000
7000
8000
Jul Aug Sep Oct Nov Dec
Month
Mathematical Functions
Y = f(x)
-0.2
0
0.2
0.4
0.6
0.8
1
-10 -8 -6 -4 -2 0 2 4 6 8 10
Scatter Plot
Time Required
0
1000
2000
3000
4000
5000
6000
7000
8000
0 10 20 30 40 50 60 70 80 90 100
Num. of Elements
Polar and Lissajous Plots
-1
-0.5
0
0.5
1
-1 -0.5 0 0.5 1
Line Plot with Impulses
0
0.5
1
-5 0 5

7

The LinePlot Widget

The LinePlot widget plots arbitrary (x,y) data points by marking each point with a small glyph and/or connecting adjacent points with lines. LinePlot resources allow you to specify the color of the markers, as well as the color, width, and line pattern of the lines. Arrays of X and Y data points are specified through two independent XiPlotData structures. This means that when data points occur at regular intervals along the X axis, you need only specify the Y data points.

LinePlot Synopsis

Class Name:	XiLinePlot
Class Hierarchy:	RectObj → XiPlot → XiLinePlot
Header File:	<*Xi/LinePlot.h*>
Class Pointer:	`xiLinePlotWidgetClass`

Constructor:	`XiCreateLinePlot(Widget parent,String name,`
	`ArgList args, Cardinal`
	`num_args);`

You can add error bars to your line plots by using the ErrorPlot widget. This is a subclass of LinePlot that is useful for displaying experimental data. The ErrorPlot subclass is documented in Chapter 8, *The ErrorPlot Widget.*

The present chapter explains how to create LinePlot widgets, documents the resources that control LinePlot appearance, and explains the functions and resources that let you specify the plot data to be displayed.

7.1 Creating a LinePlot

You can create a LinePlot widget with the function XiCreateLinePlot(), which is a standard Motif-style widget constructor. If you prefer to use XtCreateWidget() or XtVaCreateWidget(), the class pointer for the LinePlot widget is xiLinePlotWidgetClass. Both the constructor function and the widget class pointer are declared in the header file *<Xi/LinePlot.h>*.

Example 7-1 demonstrates both methods of creating the widget. Don't forget that you must manage any widgets you create with XtManageChild() or XtManageChildren().

Example 7-1. Two methods for creating a LinePlot widget.

```
#include <Xi/Plotter.h>   /* always include this */
#include <Xi/LinePlot.h> /* declares constructor and class pointer
*/

Widget plotter;  /* assume this is already created */
Widget plots[2];
Arg args[10];
int i;

/* method #1 */
i = 0;
XtSetArg(args[i], XmNlegendName, "line plot #1"); i++;
XtSetArg(args[i], XmNlineWidth, 3); i++;
XtSetArg(args[i], XmNlinePattern, XiLinePatternDotDashed2); i++;
XtSetArg(args[i], XmNdrawShadow, True); i++;
plot1 = XiCreateLinePlot(plotter, "plot1", args, i);

/* method #2 */
plot2 = XtVaCreateWidget("plot2", xiLinePlotWidgetClass, plotter,
                      XmNmarkPoints, True,
                      XmNmarker, XiMarkerCircle,
                     XtVaTypedArg,XmNmarkerColor,XmRString,"navy",5,
                      NULL);

/* the plots must be managed before they are visible */
XtManageChildren(plots, XtNumber(plots));
```

7.2 LinePlot Resources

The LinePlot widget has resources that control the line, markers, impulses, and shadows that it draws, and the data points that it plots. These resources are summarized in Table 7-1 and are documented in detail in the subsections below. For convenience, the table of resources also appears in the quick reference section at the end of this manual.

7.2.1 Lines

The following resources specify whether and how the LinePlot widget draws lines between adjacent data points.

XmNconnectPoints

A Boolean that specifies whether the LinePlot widget will connect adjacent points with lines. The default is True. If you set this resource to False, then you should generally also set XmNmarkPoints to True, or the points will not be visible at all.

XmNlineColor

The color of the line to draw between adjacent data points. If you do not set this resource when you create the LinePlot widget, it inherits the value of the XmNforeground resource of its parent Plotter widget.

XmNlineWidth

The width, in pixels, of the line to draw between adjacent data points. The default value is 0, which specifies an efficiently drawn line one-pixel wide. For many plots, a line two or three pixels wide has more visual impact, and for some presentation graphics you may want to use lines ten or more pixels wide.

XmNlinePattern

The dash pattern to use for the line between adjacent data points. The default is NULL, which specifies solid lines. Chapter 3, *Plotter Data Types and Resources* explains how to specify line patterns. A table of pre-defined line patterns appears in this chapter, and also in the quick-reference section at the end of this manual. When using line patterns, bear in mind that many X servers draw dashed and dotted lines inefficiently, and can be particularly slow when drawing patterned lines with XmNlineWidth greater than 0.

Table 7-1 LinePlot Resources

Name	Class	Type	Default	CSG	Pg
Lines					
XmNconnectPoints	XmCConnectPoints	Boolean	TRUE	CSG	111
XmNlineColor	XmCForeground	Pixel	XmNforeground[a]	CSG	111
XmNsegmentWidth	XmCLineWidth	Dimension	0	CSG	
XmNlinePattern	XmCLinePattern	LinePattern	XiLinePatternSolid	CSG	111
Markers					
XmNmarkPoints	XmCMarkPoints	Boolean	FALSE	CSG	112
XmNmarker	XmCMarker	Marker	XiMarkerCircle	CSG	112
XmNmarkerColor	XmCForeground	Pixel	XmNforeground[a]	CSG	113
XmNmarkerBackground	XmCBackground	Pixel	XmNplotAreaColor[b]	CSG	113
FillPattern					

XmNfillPattern	XmCFillPattern	FillPattern	XiFillPatternNone	CSG	114
XmNfillColor	XmCForeground	Pixel	XmNforeground[a]	CSG	114
XmNfillBackground	XmCBackground	Pixel	XmNplotAreaColor[b]	CSG	114
XmNfillOpaque	XmCFillOpaque	Boolean	TRUE	CSG	114
Impulses					
XmNdrawImpulses	XmCDrawImpulses	Boolean	FALSE	CSG	115
Shadows					
XmNdrawShadow	XmCDrawShadow	Boolean	FALSE	CSG	115
XmNshadowColor	XmCForeground	Pixel	XmNforeground[a]	CSG	115
XmNstippleShadow	XmCStippleShadow	Boolean	TRUE	CSG	116
XmNshadowXOffset	XmCShadowOffset	Position	2	CSG	116
XmNshadowYOffset	XmCShadowOffset	Position	2	CSG	116
Data[c]					
XmNnumValues	XmCNumValues	Cardinal	0	CSG	116
XmNxValues	XmCXValues	Pointer	NULL	CSG	116
XmNxValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	116
XmNxValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	116
XmNxValueOffset	XmCValueOffset	Cardinal	0	CSG	116
XmNxValueConstant	XmCValueConstant	Double	0.0	CSG	116
XmNxValueMultiplier	XmCValueMultiplier	Double	1.0	CSG	116
XmNyValues	XmCYValues	Pointer	NULL	CSG	116
XmNyValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	116
XmNyValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	116
XmNyValueOffset	XmCValueOffset	Cardinal	0	CSG	116
XmNyValueConstant	XmCValueConstant	Double	0.0	CSG	116
XmNyValueMultiplier	XmCValueMultiplier	Double	1.0	CSG	116

[a] *Inherited from the Plotter XmNforeground resource.*

[b] *Inherited form the Plotter XmNplotAreaColor resource.*

[c] *Do not set these resources when using a DBPak subclass of this widget.*

7.2.2 Markers

The following resources specify whether and how the LinePlot widget marks each of its data points.

XmNmarkPoints

A Boolean resource that specifies whether the LinePlot widget should place a small glyph to mark each data point. The default is False.

XmNmarker

The XiMarker that the LinePlot widget should use to mark its data points. The default is the predefined value XiMarkerCircle. Chapter 3, *Plotter Data Types and Resources* explains the XiMarker abstraction and contains a list of predefined markers that you can use for this resource. This list also appears in the quick reference section at the end of this manual. Chapter 17, *Defining Custom*

Markers explains how you can define your own custom marker glyphs to display in X and print in PostScript.

XmNmarkerColor

The foreground color for the marker. If you do not set this resource when the LinePlot widget is created, its value is inherited from the XmNforeground resource of the parent Plotter widget.

XmNmarkerBackground

The background color for the marker. If you do not set this resource, its value is inherited from the XmNplotAreaColor resource of the parent Plotter widget. The background color is used only for those marker types that are unfilled, and have a "hollow" section in the center. Usually, you will want this resource to match the Plotter XmNplotAreaColor, as it does by default, so that the markers actually do appear hollow. You can use this background resource, however, to create two-color markers; note, though, that the background color will not be used when the LinePlot is printed. Finally, note that this resource is not affected by changes to the Plotter XmNplotAreaColor resource. If you change the background color of the plotting area without changing the background color of the markers for each of your LinePlot widgets, markers that previously appeared hollow will become two-tone.

7.2.3 Fill Patterns

The LinePlot widget can optionally fill the area between its data points and the origin (or between the data and the bottom or top edge of the Plotter, if the origin is not visible) with a solid color or with a pattern. To have it fill this area, set the XmNfillPattern resource to specify the type of fill to perform and the XmNfillColor resource to specify the color. You may also want to specify XmNfillBackground and XmNfillOpaque to specify how the background color, if any, should be handled.

Note that displaying a LinePlot in this way may not look good unless XmNconnectPoints is True. If you have more than one LinePlot widget that you wish to fill beneath, you must carefully set their drawing order (see the Plotter XmNdrawingOrder constraint) so that the smaller filled areas are drawn on top of the larger filled areas, instead of being obscured by those larger filled areas.

The resources that control filling in the LinePlot are the following:

XmNfillPattern

The pattern to be used to fill beneath the LinePlot data. The default is the special value XiFillPatternNone which specifies that no filling should occur. The value XiFillPatternSolid specifies that the area beneath the data points should be filled with a solid color specified by XmNfillColor. Other values specify a patterned fill.Chapter 3, *Plotter Data Types and Resources* explains the XiFillPattern abstraction, and that chapter and the quick reference section at the end of this manual contain lists of the predefined fill patterns. You can also define custom fill patterns of your own; Chapter 18, *Defining Custom Fill Patterns* explains how to do this.

XmNfillColor

The solid color to fill with, or the foreground color of the fill pattern. If you do not set this resource, its value is inherited from the XmNforeground resource of the parent Plotter widget.

XmNfillBackground

The background color of the pattern used to fill with. If XmNfillPattern is XiFillPatternSolid, or XmNfillOpaque is False, then this resource is not used. If you do not set this resource, its value is inherited from the XmNplotAreaColor resource of the parent Plotter widget. This default value will make a patterned fill appear to have a transparent background.

XmNfillOpaque

A Boolean that specifies whether pattern fills should be "opaque" or "transparent". With an opaque fill, the LinePlot widget draws both the foreground and background bits of the pattern, so that everything in the area beneath the data is erased. This is the default, and is almost always the appropriate setting for this resource. If you set it to False, then only the foreground bits of the fill pattern will be drawn, and anything previously drawn underneath the data (such as grid lines) will show through.

7.2.4 Impulses

Besides connecting adjacent points and placing a marker at each point, the LinePlot widget can draw an "impulse" for each point. An impulse is a line drawn vertically from the origin (or the minimum value of the Y axis) to the data point. Impulses can be used alone, and also in conjunction with markers and/or lines. When displaying sampled data, in signal processing contexts, for example, using impulses can help to emphasize the sampled nature of the data. A single resource controls whether or not the LinePlot draws impulses:

XmNdrawImpulses

A Boolean that specifies whether impulse lines should be drawn vertically from the origin of the Y axis, or from the minimum value on the Y axis, if the Y axis does not pass through 0.0, to the data point. The default value is False.

Impulse lines are drawn using the same attributes as the connecting lines in the line plot: XmNlineColor, XmNlineWidth, and XmNlinePattern. Impulse lines are never shadowed, so you probably do not want to set both XmNdrawImpulses and XmNdrawShadow to True.

7.2.5 Shadows

The LinePlot widget can draw a shadow beneath the markers it draws at data points, and the lines it draws between those points. This shadow gives the plot a somewhat three-dimensional appearance and can add dramatically to the visual impact of the plot. LinePlot shadows are useful for presentation graphics, but generally should not be used for scientific or engineering graphs, because they distract attention from the underlying data that the plot represents.

The following resources control whether and how the LinePlot draws its shadows:

XmNdrawShadow

A Boolean that specifies whether the LinePlot widget will draw shadows beneath its markers and lines. The default is False.

XmNshadowColor

The color of the shadow to be drawn. If you do not set this resource, its value is inherited from the XmNforeground resource of the parent Plotter widget. Note that if XmNstippleShadow is True, then the stippling will result in an apparent shadow color that is significantly lighter than the value specified by this resource.

XmNstippleShadow

A Boolean that specifies whether the LinePlot shadows should be drawn with a solid color or with a stippled color. The default is True which specifies that a stipple should be used. Note that on monochrome systems, stippling is required because the shadow color and the line color will be the same. Note also that shadows will always be stippled in PostScript output, regardless of the state of this resource.

XmNshadowXOffset, XmNshadowYOffset

The offset, in pixels, of the shadow from the line plot in the X and Y dimensions. The default for each of these resources is 2 pixels. Larger values will produce a shadow further away from the actual line plot, increasing the apparent three dimensional height of the LinePlot above the Plotter background. If you are using particularly wide lines in a LinePlot, you may want to increase this shadow offset as well. Note that negative values are legal for this resource, and simply change the position of the imaginary light source that is casting the shadow.

7.2.6 Data Resources

Table 7-1 lists thirteen data resources for the LinePlot widget. These resources correspond to the fields of two XiPlotData structures which specify how the LinePlot widget is to extract its X and Y data values from application data structures. These resources are not explained here; if you have read Chapter 4, *Plot Data Representation*, the meaning of each of them will be evident from their names.

Note that there is only one XmNnumValues resource, rather than separate XmNx-NumValues and XmNyNumValues resources--since there must always be the same number of X and Y points, the LinePlot only needs one resource to set the appropriate fields in both XiPlotData structures. You must be careful when setting this XmNnumValues resource to ensure that the arrays specified on the XmNxValues and XmNyValues resources have at least that many elements. If you increase the size of XmNnumValues before you update XmNyValues to point to its new, longer, array, the LinePlot widget may attempt to read data beyond the end of the array, which can cause a segmentation violation.

The LinePlot widget is often used to display graphs of data that has been measured, sampled, or computed at some regular interval along the X axis. In this case, there is no need to provide a complete array of X points; instead, you need only specify the first X point, and the interval between points on the X axis. You can do this with the XmNxValueConstant and the XmNxValueMultiplier resources. Again, see Chapter 3 for more information on these resources. This technique will obviously not work when plotting polar data, parametric functions, scatter plots, or any kind of data that does not have evenly spaced X points.

Finally, note that the convenience functions for specifying data that are described in the next section are often easier to use (though they are not quite so flexible) than these resources are.

7.3 Specifying LinePlot Data

The most general way to specify data for a LinePlot widget is through the data resources described in the previous section. It is often easier, however, to use one of the convenience functions provided by the LinePlot for this purpose. The subsections below document these convenience functions.

7.3.1 Arrays of Doubles, Floats, or Ints

The simplest of the LinePlot convenience functions allow you to specify X and Y data stored in separate arrays of type double, float, or int. These functions are shown in Figure 7-1.

XiLinePlotSetDoublePoints(), XiLinePlotSetFloatPoints(), and XiLinePlotSetIntPoints() are appropriate when you want to specify arrays of both X and Y data. But many line plots have evenly spaced X data that can be fully specified with only the minimum value and the interval between values. In this case, you can use XiLinePlotSetDoubleYValues, XiLinePlotSetFloatYValues(), or XiLinePlotSetIntYValues(). These three functions they are very similar to the previous three, except that they take X minimum and X interval arguments instead of an array of X values.

7.3.2 Specifying Data in Structures

Sometimes you may want to plot data that is stored in an array of structures, rather than a simple array of double, float, or int. Or you may want to plot data of some other type, like unsigned short. The LinePlot widget has two convenience functions that allow this--one used when you want to specify X and Y points, and one when you only want to specify Y values. Figure 7-2 shows these functions.

```
void XiLinePlotSetDoublePoints(Widget w,
                              double *xvals,double *yvals,Cardinal num);
void XiLinePlotSetFloatPoints(Widget w,
                              float *xvals, float *yvals, Cardinal num);
void XiLinePlotSetIntPoints(Widget w,int*xvals,int*yvals,Cardinal num);

void XiLinePlotSetDoubleYValues(Widget w, double xmin,double xinterval,
                                 double *yvals, Cardinal num);
void XiLinePlotSetFloatYValues(Widget w, double xmin, double xinterval,
                                float *yvals, Cardinal num);
void XiLinePlotSetIntYValues(Widget w, double xmin, double xinterval,
                              int *yvals, Cardinal num);
```

`w` The LinePlot widget that is having its data values set.

`xvals, yvals`

Arrays of double, float, or Wint which contain the X and Y data values to be plotted.

`xmin` The X coordinate of the first point in the plot.

`xinterval`

The interval between adjacent points along the X axis.

`num`

The number of points in the plot. There must be at least this many elements in each of the arrays.

Figure 7-1. Functions for specifying (X,Y) LinePlot data in simple arrays.

The arguments to XiLinePlotSetPoints() correspond to the XmNxValue resources, the XmNyValue resources, and the XmNnumValues resource of the LinePlot widget. Note, though that there are no arguments that correspond to the "constant" and "multiplier" resources. Except for these linear transform resources, however, XiLinePlotSetPoints() provides all the flexibility of the widget data resources. Since this function requires you to specify a complete array of X values, you would use it for scatter plots, parametric functions, or other plots that do not have a uniform point spacing along the X axis.

XiLinePlotSetYValues() is similar to XiLinePlotSetPoints(), but lets you specify your X data with a minimum value and an interval, rather than a complete array of values. The xmin and xinterval arguments are the equivalent of the XmNxValueConstant and XmNxValueMultiplier resources. You would use this function whenever you are plotting sampled data, or any plot with a uniform interval between X points.

```
void XiLinePlotSetPoints(Widget w,XtPointer xvals,XiPlotDataType xtype,
                          Cardinal xsize, Cardinal xoffset,
                          XtPointer yvals, XiPlotDataType ytype,
                        Cardinal ysize, Cardinal yoffset, Cardinal num);
void XiLinePlotSetYValues(Widget w, double xmin, double xinterval,
                           XtPointer yvals, XiPlotDataType ytype,
                         Cardinal ysize, Cardinal yoffset,Cardinal num);
```

`w`

The LinePlot widget that is having its data set.

`xvals, xtype, xsize, xoffset`

An array of structures containing the X values, the type of each value, the size of each array element, and the offset of the X value within the structure. These arguments correspond to the X data resources.

`yvals, ytype, ysize, yoffset`

An array of structures containing the Y values, the type of each value, the size of each array element, and the offset of the Y value within the structure. These arguments correspond to the Y data resources.

`xmin, xinterval`

The X coordinate of the first point in the plot, and the interval between adjacent X points. This is an alternate way of specifying X data.

`num`

The number of points in the plot. Both the xvals and the yvals arrays must have at least this many elements.

Figure 7-2. Functions for specifying LinePlot data in structures

7.3.3 Changing Data in Place

Since the LinePlot widget extracts its data values directly from application data structures, it needs to be notified when that data has changed "underneath" it. Some applications will define static data structures, and leave them unchanged for the lifetime of the program. Other applications, however, will update plot data dynamically. One way to do this is to give the widget a pointer to new data arrays (by setting resources or calling one of the convenience functions above), and then free the memory associated with the old ones. Another approach is to write the new plot data directly into the existing arrays, and notify the widget that its data has changed and that it should redraw itself. You can do this with XiLinePlotDataChanged(), shown in Figure 7-3.

```
void XiLinePlotDataChanged(Widget w);
```

`w` The LinePlot widget that has had its data changed. Calling this function will cause that widget to extract and display new data from the application data structures.

Figure 7-3. XiLinePlotDataChanged

7.3.4 Adding Data Points Dynamically

Some applications will want to use the Plotter widget to display "live" data--to update data dynamically as it is computed, generated, or otherwise becomes available. One way to do this is to simply update the widget's data resources, or call one of its convenience functions for setting data. This approach will cause the widget to reread, rescale and redisplay all of its data values, and is not a particularly efficient technique if new data points are only being added at the end of the plot.

Since the LinePlot widget is the most commonly used with this kind of live data, it supports a special convenience function, XiLinePlotDataAdded(), that allows you to efficiently add new data points to the end of a LinePlot. Figure 7-4 shows the signature of this function.

```
void XiLinePlotDataAdded(Widget w, Cardinal new_num);
```

`w` The LinePlot widget that is having data points added.

`new_num`

The new number of points in the LinePlot data arrays. Calling this function will add the new points to the end of the plot without redrawing the existing points.

Figure 7-4. XiLinePlotDataAdded()

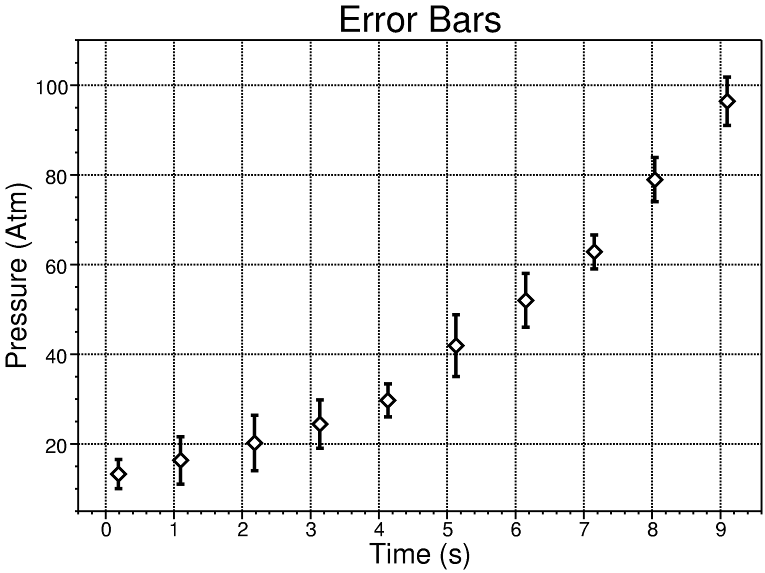
You use XiLinePlotDataAdded(), like XiLinePlotDataChanged(), when you have updated the LinePlot data "in place"--that is, when the data has changed, but the pointers to the array or arrays of data have remained the same. XiLinePlotDataAdded() takes the new number of data points as its second argument. It assumes that none of the existing data has changed, and only reads, scales, and draws the new data points. This generally results in a fast update. Note, however, that if autoscaling is on, adding new points to the plot can cause the axes to be rescaled, if any of the new points fall outside of the current axis bounds. When this happens, all of the data displayed in the Plotter will have to be rescaled, and the update will be a slow one.

Example 7-2. Using XiLinePlotDataAdded()

```
static int ypoints[50];
static int num_points = 0;
static double xmin = 0.0;
static double xmax = 50.0;

void AddPoint(int new_y_value)
{
    /* add the new value to our array */
    ypoints[num_points++] = new_y_value;

    /*
     * if we've filled up the array, throw out the first 25 points,
     * adjust the xaxis and the plot, and redisplay the plot.
     * Otherwise, just add the point to the plot efficiently.
     */
    if (num_points == 50) {
        bcopy(&ypoints[25], ypoints, 25 * sizeof(int));
        num_points = 25;
        xmin += 25.0;
        xmax += 25.0;
        XiPlotterDisableRedisplay(plotter);
        XiAxisSetBounds(XiPlotterXAxis(plotter), xmin, xmax);
        XiLinePlotSetIntYValues(plot1, xmin, 1.0, ypoints,
num_points);
        XiPlotterEnableRedisplay(plotter);
    }
    else
        XiLinePlotDataAdded(plot1, num_points);
}
```

When dynamically updating data with XiLinePlotDataAdded(), the X axis will generally measure time, or some other monotonically increasing quantity. When this is the case, you will want to be sure to turn off autoscaling on the X axis, or the axis will rescale itself (and all data in the Plotter) each time you add a point. Instead you will have to explicitly specify a minimum and maximum bound for the X axis. As you add data points, the line plot will eventually reach the maximum value you have specified. If you continue to add points, they will not appear because they will be off the edge of the plotting area. Instead, the behavior you often want is to scroll the line plot to the left each time it reaches the right edge of the plotting area. This is the default behavior of the xload application, for example. Figure 7-2 shows a procedure you might use to do this.

The procedure shown in the example simply discards values once they have been scrolled off the left of the plotting area. Another approach is to retain the data points, and allow the user to manually scroll backwards to view past data. You might use the Plotter XmNpanCallback to allow the user to scroll.


Error Bars
Pressure (Atm)
100
80
60
40
20
0
1
2
3
4
5
6
7
8
9
Time (s)

8

The ErrorPlot Widget

The ErrorPlot widget is a subclass of the LinePlot widget which displays error bars for each data point of a plot--this is useful for scientific and engineering data, and any kind of plot for which there is uncertainty in the measurement of data. Because the ErrorPlot is a subclass of LinePlot, it inherits all the resources of that widget. This means that you can specify data points, how those points are to be marked, how they are to be connected, and so on, exactly as you would for a LinePlot. (See Chapter 7, *The LinePlot Widget* for information on that widget.)

ErrorPlot Synopsis

Class Name:	XiErrorPlot
Class Hierarchy:	RectObj → XiPlot → XiErrorPlot
Header File:	<*Xi/ErrorPlot.h*>
Class Pointer:	`xiErrorPlotWidgetClass`

Constructor:	`xiCreateErrorPlot (Widget parent,String name,`
	`ArgList args, Cardinal num_args);`

Error values can be specified for an ErrorPlot in three different ways: If there is a fixed error for all points in a plot, you can specify it easily as a single absolute error value above and below each data point. Similarly, if there is a single percentage error for all measured values in a plot, you can specify this as a single value. Finally, if errors vary from point to point, you can explicitly specify the bounds of the error bar above and below each point in the plot.

Error bars appear like the capital letter 'I'. Resources allow you to control the color of the bar, the width of the lines used to draw the bar, and the length of the "caps"--the horizontal lines at the top and bottom of the bar.

This chapter explains how to create ErrorPlot widgets, documents the resources that control ErrorPlot appearance, and explains how to specify error values for a plot.

8.1 Creating an ErrorPlot

You can create an ErrorPlot widget with the function XiCreateErrorPlot(), which is a standard Motif-style widget constructor, or with XtCreateWidget() or XtVaCreateWidget() with the widget class pointer xiErrorPlotWidgetClass. Both the constructor function and the widget class pointer are declared in the header file <*Xi/ErrorPlot.h*>.

Example 8-1 demonstrates both methods of creating the widget

Example 8-1. Creating ErrorPlot widgets

```
#include <Xi/Plotter.h>    /* always include this */
#include <Xi/ErrorPlot.h>  /* declares constructor and class pointer
*/

Widget plotter;  /* assume this is already created */
Widget plots[2];
Arg args[10];
int i;

/* method #1 */
i = 0;
XtSetArg(args[i], XmNerrorBarWidth, 3); i++;
XtSetArg(args[i], XmNlineWidth, 3); i++;
XtSetArg(args[i], XmNerrorCapLength, 5); i++;
XtSetArg(args[i], XmNmarkPoints, True); i++;
plot1 = XiCreateErrorPlot(plotter, "plot1", args, i);

/* method #2 */
plot2 = XtVaCreateWidget("plot2", xiErrorPlotWidgetClass, plotter,
                     XmNmarkPoints, True,
                     XmNmarker, XiMarkerSquare,
                     XtVaTypedArg,XmNlineColor,XmRString,"tan",4,
                    XtVaTypedArg,XmNerrorBarColor,XmRString,"tan",4,
                     NULL);

/* the plots must be managed before they are visible */
XtManageChildren(plots, XtNumber(plots));
```

8.2 ErrorPlot Resources

The ErrorPlot widget inherits all the resources of the LinePlot widget. It also defines three new resources that control the appearance of the error bars, and a number of other resources that allow you to specify error values for a plot. These new resources are summarized in Table 8-1 and are documented in the subsections below. Chapter 7, *The LinePlot Widget* documents the resources of the LinePlot widget; you will need to use these resources to specify how data points should be marked, connected, and so on. For convenience tables of LinePlot and ErrorPlot resources appear in the quick reference section at the end of this manual.

Table 8-1. ErrorPlot Resources

Name	Class	Type	Default	CSG	Pg
Error Bars					
XmNerrorBarColor	XmCForeground	Pixel	XmNforeground[a]	CSG	127
XmNerrorBarWidth	XmCLineWidth	Dimension	0	CSG	127
XmNerrorCapLength	XmCErrorCapLength	Dimension	3	CSG	127
Fixed Errors					
XmNfixedError	XmCFixedError	Double	0.0	CSG	128
XmNpercentError	XmCPercentError	Double	0.0	CSG	128
Variable Errors[b]					
XmNnumErrorValues	XmCNumValues	Cardinal	0	CSG	129
XmNhighValues	XmCValues	Pointer	NULL	CSG	129
XmNhighValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	129
XmNhighValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	129
XmNhighValueOffset	XmCValueOffset	Cardinal	0	CSG	129
XmNlowValues	XmCValues	Pointer	NULL	CSG	129
XmNlowValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	129
XmNlowValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	129
XmNlowValueOffset	XmCValueOffset	Cardinal	0	CSG	129

[a] *Inherited from the Plotter XmNforeground resource.*

[b] *Do not set these resources when using a DBPak subclass of this widget.*

8.2.1 Error Bar Resources

An error bar looks like a capital letter 'I'. The following resources control the color of the bar, the width of the lines that comprise it, and the length of the horizontal lines at the top and bottom of the bar.

XmNerrorBarColor

The color of the error bar. If no value is specified for this resource, the default value is inherited from the XmNforeground resource of the parent Plotter widget.

XmNerrorBarWidth

The width, in pixels, of the lines that comprise the error bar. The default is 0, a special value that specifies efficiently drawn lines one pixel wide.

XmNerrorCapLength

The length, in pixels, from the center of an error bar to the left and right end of the horizontal line segments at the top and bottom of the bar. The default is 3, which specifies a cap with a total length of 7 pixels (the center pixel of the bar plus three pixels on the left and three on the right). When using a wide value for XmNerrorBarWidth, you may want to increase the value of XmNerrorCapLength, because the wide vertical line in the center will hide some of the length of the cap.

Note that there are not resources for specifying shadows for an error bar. Because the ErrorPlot widget is used in situations where accurate representation of data is very important, the widget does not support shadows underneath error bars--they would only distract from the actual data points and the measurement errors being plotted. Thus, when using the ErrorPlot widget, you should leave the XmNdrawShadow LinePlot widget at its default value of False, or there will be shadows under the plot lines and markers but not under the error bars.

8.2.2 Fixed Error Resources

Some plots will have a fixed error across all points in the data set. You might be plotting weights which have been measured with an accuracy of (+-5g, for example, or electrical resistances with a rated tolerance of (+-10%. If you have a fixed error value (either an absolute error value or a percentage) like this, then you specifying the error values for your plot is simple; you need only set one of the two following resources:

XmNfixedError

An absolute error value above and below each point in the plot. Use this resource when your errors are of the form (+-.05m, for example. The default is 0.0, which means that no fixed error is specified. Note that this resource is ignored if XmNnumErrorValues is set; in that case high and low error values are looked up for each point. Also, remember that this resource if of type double, and special techniques are required to set it. (See Chapter 3, *Plotter Data Types and Resources.*)

XmNpercentError

A percentage error value above and below each point in the plot. Use this resource when your errors are of the form \(+-10%, for example. Because the error is a percentage of the data value, values close to zero will have smaller error bars than those farther away. The default is 0.0, which means that no percentage error is specified. Note that this resource is ignored if XmNnumErrorValues or XmNfixedError is specified. Also, remember that this resource is of type double, and special care is required when setting it.

Notice that there are not ErrorPlot resources for specifying the plot data values themselves, only the errors for those data values. Since the ErrorPlot is a subclass of LinePlot, you specify the plot data through inherited LinePlot resources or convenience functions.

8.2.3 Variable Error Resources

When your data does not have the same fixed error for each point, the ErrorPlot widget allows you to specify the absolute high and low bound of the error bar for each data point in the plot. Table 8-1 lists nine resources that allow you to specify these high and low error values. These resources are the fields of two XiPlotData structures that tell the ErrorPlot widget how to extract its high and low error values from application data structures. Note that the high and low values in these data structures absolute values, not \(+- values relative to the data Y value. The resources are not individually explained here; if you have read Chapter 4, *Plot Data Representation* the meaning of each of them will be evident from their names. The ErrorPlot widget also provides a number of convenience functions for specifying error values; these functions are described in the next section, and are often easier to use than these resources.

Note that there is only a single XmNnumErrorValues resource, rather than individual XmNhighNumValues and XmNlowNumValues resources--since there must always be the same number of high and low error values, only one resource is necessary to specify the number of values in both XiPlotData structures. You must be careful when setting the XmNnumErrorValues resource to ensure that there are at least that many values in the XmNhighValues and XmNlowValues arrays; if there are not, the ErrorPlot widget may read beyond the end of the arrays and cause a segmentation violation.

The ErrorPlot high and low data resources specify only the values to be displayed by the error bar for each point; they do not specify the position of the points themselves--this remains the job of the LinePlot (LinePlot is the superclass) X and Y data resources and convenience functions. See Chapter 7, *The LinePlot Widget* for more information.

Finally, notice that the ErrorPlot widget does not have resources to set the constant and multiplier fields of the error value XiPlotData structures. Since there is probably no realistic case where error values will be linear, this is not really a problem. It does mean that although you can apply a linear transformation to the data values of an ErrorPlot, you cannot automatically apply the same transform to the error values.

8.3 Specifying ErrorPlot Data

The most general way to specify high and low error values for an ErrorPlot widget is through the data resources described in the previous section. When your data is stored in certain common formats, however, it is often easier to use one of the convenience functions provided by the ErrorPlot for this purpose. The subsections below document these convenience functions.

8.3.1 Arrays of Doubles, Floats, or Ints

The simplest of the ErrorPlot convenience functions allow you to specify high and low error values that are stored in separate arrays of type double, float, or int. These functions are shown in Figure 8-1

Notice that these functions only specify the error values for an ErrorPlot. Before anything can be displayed, you must also specify the data points for each plot. You can do this with LinePlot functions like XiLinePlotSetDoubleValues(), or through LinePlot data resources.

8.3.2 Specifying Error Values from Structures

Often, it is more convenient for an application to store its pairs of high, low error values together in the same structure, perhaps in conjunction with the (X,Y) coordinates of the data point itself. If you store your error values in this way, you can use XiErrorPlotSetErrors() to specify the error values. This arguments to this function are shown in Figure 8-2.

```
void XiErrorPlotSetDoubleErrors(Widget w, double* highvals,
                                 double *lowvals,
                                 Cardinal num_values)
void XiErrorPlotSetFloatErrors(Widget w,float *highvals,float *lowvals,
                                Cardinal num_values);
void XiErrorPlotSetIntErrors(Widget w, int *highvals, int *lowvals,
                              Cardinal num_values);
```

`w` The ErrorPlot widget that is having its error values set.

`highvals, lowvals`

Two arrays of, of type double, float, or int, depending on the function, which contain the high and low values for each error bar in the plot.

`num_values`

The number of error bars to appear in the plot. There must be at least this many values in the highvals and lowvals arrays.

Figure 8-1. ErrorPlot functions for specifying error values in simple arrays

```
void XiErrorPlotSetErrors(Widget w,
                           XiPlotDataType type, Cardinal size,
                           XtPointer highvals, Cardinal highoffset,
                           XtPointer lowvals, Cardinal lowoffset,
                           Cardinal num_values);
```

`w` The ErrorPlot widget that is having its error values set.

`type` The type of both the high and low error values.

`size` The size of the structure that both the high and low values are stored in.

`highvals, lowvals`

Pointers to arrays of structures that contain the high and low values. Often these two arguments will be the same.

`highoffset, lowoffset`

The offset, in bytes, of the high value and the low value within the structure that they are stored in.

`num_values`

The number of error bars in the plot. The highvals and lowvals arrays must each have at least this many elements.

Figure 8-2. XiErrorPlotSetErrors()

Again, note that this function sets only the error values for a plot; to set the data points themselves, use the superclass LinePlot convenience function or data resources. Example 8-2 shows how you might use XiErrorPlotSetErrors() to set error values that are stored in an array of structures.

Example 8-2. Specifying error values from structures

```
typedef struct {
    float y;          /* the y coordinate of the data point */
    float high, low;  /* the y coordinates of the error bar */
} Point;

static Point points[50];  /* assume this is initialized elsewhere */

/* set data points with a LinePlot function */
XiLinePlotSetYValues(errorplot, 0.0, 1.0,
                      points, XiPlotDataFloat,
                      sizeof(Point), XtOffsetOf(Point, y),
                      XtNumber(points));
/* and then set error values */
XiErrorPlotSetErrors(errorplot,
                      XiPlotDataFloat, sizeof(Point),
                      points, XtOffsetOf(Point, high),
                      points, XtOffsetOf(Point, low),
                      XtNumber(points));
```

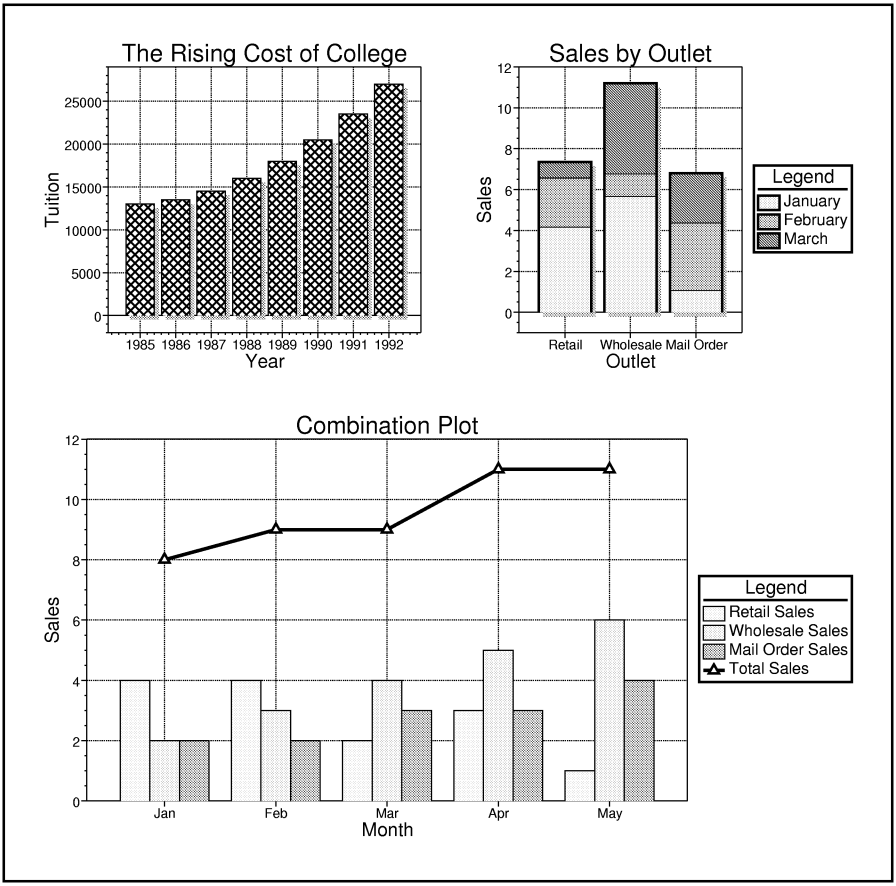
8.3.3 Changing Data in Place

To change the error values of an ErrorPlot, you can pass a new array to the widget with XiErrorPlotSetErrors() or a related function, or, if the number of points remains the same, you can simply change the values in the existing array. If you take this latter course, you must notify the ErrorPlot widget of the change by calling XiErrorPlotDataChanged(), because the widget will have no other way of detecting it.

```
void XiErrorPlotDataChanged(Widget w);
```

`w` The ErrorPlot widget that has had its data changed. Calling this function will cause that widget to extract and display new error values from the application data structures.

Note also that if the data point value changed at the same time as the error values, you will probably also want to call XiLinePlotDataChanged().


The Rising Cost of College
Tuition
0
5000
10000
15000
20000
25000
1985 1986 1987 1988 1989 1990 1991 1992
Year
Sales by Outlet
Sales
0
2
4
6
8
10
12
Retail Wholesale Mail Order
Outlet
Legend
January
February
March
Combination Plot
Sales
0
2
4
6
8
10
12
Jan
Feb
Mar
Apr
May
Month
Legend
Retail Sales
Wholesale Sales
Mail Order Sales
Total Sales

9

The BarPlot Widget

The BarPlot widget displays data by plotting vertical rectangles between the origin of the Y axis and the Y data point. Resources allow you to specify a fill pattern and a fill color for the bar, as well as the color and width of the lines used to outline the bar. Other resources control the drawing of shadows for each bar. You specify only the Y coordinate of each data point; bars are assumed to be evenly spaced in the X dimension and there are resources that let you specify the first X value and the interval between values on the X axis. There are resources to specify the width of each bar, and an offset to the left or right of the X data point. Multiple BarPlot widgets in a Plotter can automatically cluster themselves into groups at each X data point, or can automatically stack themselves at each point.

BarPlot Synopsis

Class Name: XiBarPlot

Class Hierarchy: RectObj → XiPlot → XiBarPlot

Header File: *<Xi/BarPlot.h>*

Class Pointer: xiBarPlotWidgetClass

Constructor:

```
XiCreateBarPlot (Widget parent, String name,
                 ArgList args,
                 Cardinal num_args);
```

This chapter explains how to create BarPlot widgets, documents the resources that control BarPlot appearance, grouping and stacking, and explains the functions and resources that specify the data to be plotted.

9.1 Creating a BarPlot

You can create a BarPlot widget with the function XiCreateBarPlot(), which is a standard Motif-style widget constructor. If you prefer to use XtCreateWidget() or XtVaCreateWidget(), the class pointer for the BarPlot widget is xiBarPlotWidgetClass. Both the constructor function and the class pointer are declared in the header file *<Xi/BarPlot.h>*.

Example 9-1 demonstrates both methods of creating the widget. Don't forget that you must manage any widgets you create with XtManageChild() or XtManageChildren().

Example 9-1. Creating BarPlot Widgets

```
#include <Xi/Plotter.h>  /* always include this */
#include <Xi/BarPlot.h>  /* declares constructor and class pointer*/

Widget plotter;  /* assume this is already created */
Widget bar1, bar2;
Arg args[10];
int i;

XiPlotterDisableRedisplay(plotter);  /* don't refresh for now */

/* XiCreateBarPlot() method */
i = 0;
XtSetArg(args[i], XmNlegendName, "Widget Sales"); i++;
XtSetArg(args[i], XmNfillPattern, XiFillPatternGray1);
bar1 = XiCreateBarPlot(plotter, "widgets", args, i);
XtManageChild(bar1);

/* class pointer method */
XtVaCreateManagedWidget("gadgets", xiBarPlotWidgetClass, plotter,
                        XmNlegendName, "Gadget Sales",
                        XmNfillPattern, XiFillPatternGray3,
                        NULL);
XiPlotterEnableRedisplay(plotter);  /* turn updating back on */
```

9.2 BarPlot Resources

The BarPlot widget has resources that control its fill color and fill pattern, line color and width, shadows, bar grouping and stacking, and the data points that it plots. These resources are summarized in Table 9-1 and are documented in detail in the subsections below. For convenience, Table 9-1 also appears in the quick reference section at the end of this manual.

9.2.1 Bar Color and Pattern

The BarPlot widget fills its bars with a solid color by default. You can specify this color, and can also specify a fill pattern to use instead of a solid color. The following resources control how the bars are filled:

XmNfillColor

The solid color to fill the bars with, or the foreground color of the fill pattern. If you do not set this resource, its value is inherited from the XmNforeground resource of the parent Plotter widget.

XmNfillBackground

The background color of the pattern used to fill the bars of the BarPlot. If XmNfillPattern is XiFillPatternSolid, or XmNfillOpaque is False, then this resource is not used. If you do not set this resource, its value is inherited from the XmNplotAreaColor resource of the parent Plotter widget. This default value will make patterned bars appear transparent.

XmNfillPattern

The pattern to be used to fill each bar of the BarPlot. The default is the special value XiFillPatternSolid which specifies bars should be drawn with a solid color. Chapter 3, *Plotter Data Types and Resources* explains the XiFillPattern abstraction, and that chapter and the quick reference section at the end of this manual contain lists of the predefined fill patterns. You can also define custom fill patterns of your own; Chapter 18, *Defining Custom Fill Patterns* explains how to do this.

XmNfillOpaque

A Boolean that specifies whether pattern fills should be "opaque" or "transparent". With an opaque fill, the BarPlot widget draws both the foreground and background bits of the pattern, so that anything under the bars is erased. This is the default, and is almost always the appropriate setting for this resource. If you set it to False, then only the foreground bits of the fill pattern will be drawn, and anything previously drawn underneath the bars (such as grid lines) will show through.

Table 9-1. BarPlot Resources

Name	Class	Type	Default	CSG	Pg
FillPattern					
XmNfillColor	XmCForeground	Pixel	XmNforeground[a]	CSG	137
XmNfillBackground	XmCBackground	Pixel	XmNplotAreaColor[b]	CSG	137
XmNfillPattern	XmCFillPattern	FillPattern	XiFillPatternSolid	CSG	137
XmNfillOpaque	XmCFillOpaque	Boolean	TRUE	CSG	137
Lines					
XmNlineColor	XmCForeground	Pixel	XmNforeground[a]	CSG	139
XmNlineWidth	XmCLineWidth	Dimension	0	CSG	139
Shadows					
XmNdrawShadow	XmCDrawShadow	Boolean	FALSE	CSG	139
XmNshadowColor	XmCForeground	Pixel	XmNforeground[a]	CSG	139
XmNshadowPattern	XmCShadowPattern	FillPattern	XiFillPatternGray3	CSG	139
XmNshadowXOffset	XmCShadowOffset	Position	4	CSG	139
XmNshadowYOffset	XmCShadowOffset	Position	4	CSG	139
Bars					
XmNbarSize	XmCBarSize	Short	80	CSG	142
XmNbarPosition	XmCBarPosition	Short	0	CSG	142
XmNstacked	XmCStacked	Boolean	FALSE	CG	142
XmNorigin	XmCOrigin	Double	0.0	CSG	142
Data[c]					
XmNnumValues	XmCNumValues	Cardinal	0	CSG	143
XmNxMin	XmCXMin	Double	1.0	CSG	143
XmNxInterval	XmCXInterval	Double	1.0	CSG	143
XmNvalues	XmCValues	Pointer	NULL	CSG	143
XmNvalueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	143
XmNvalueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	143
XmNvalueOffset	XmCValueOffset	Cardinal	0	CSG	143
XmNvalueConstant	XmCValueConstant	Double	0.0	CSG	143
XmNvalueMultiplier	XmCValueMultiplier	Double	1.0	CSG	143

[a] *Inherited from the Plotter XmNforeground resource.*

[b] *Inherited from the Plotter XmNplotAreaColor resource.*

[c] *Do not set these resources when using a DBPak subclass of this widget.*

9.2.2 Bar Outline

The BarPlot widget always outlines each bar with a line drawn on the left, top, and right sides of the bar. (The X axis, or the origin line drawn by the Y axis serves as a the bounding line for the bottom of each bar.) The following two resources set the color and width of this outline:

XmNlineColor

The color of the bar outline. If you do not set this resource, its value is inherited from the XmNforeground resource of the parent Plotter widget.

XmNlineWidth

The width, in pixels, of the outline drawn around each bar. The default is 0, which is a special value that specifies efficiently-drawn lines one pixel wide.

9.2.3 Bar Shadows

Like the LinePlot widget, the BarPlot can draw shadows underneath each of its bars. These shadows give BarPlots a three-dimensional effect that is visually striking for presentation graphics. For scientific and engineering data, you should probably refrain from using shadows, because they distract from the actual data that is being displayed.

XmNdrawShadow

A Boolean that specifies whether shadows should be drawn under the bars of this BarPlot. The default is False.

XmNshadowColor

The color of the shadow to draw under each plot. If you do not set this resource, its value is inherited from the XmNforeground resource of the parent Plotter widget. Note that when this shadow color is combined with the stipple pattern specified by XmNshadowPattern, the apparent shadow color that results can be significantly lighter than the shadow color alone would be.

XmNshadowPattern

The fill pattern to use as a stipple for the shadow. The default is XiFillPatternGray3, which is a 50% gray pattern. The gray fill patterns all make good shadow stipples, but many of the other pre-defined patterns are not appropriate for this use.

XmNshadowXOffset, XmNshadowYOffset

The X and Y offsets, in pixels, of the bars' shadows from the bars themselves. The default for each of these resources is 2 pixels. Larger values increase the separation between bar and shadow, making the bar appear three-dimensionally further above the plotter background. Negative values are legal for both of these resources; varying their values moves the position of the imaginary light source that casts the shadows.

9.2.4 Bar Positioning, Grouping and Stacking

The BarPlot widget requires its bars to be evenly spaced along the X axis. By default, it centers its bars over each data point on the X axis, and makes them wide enough to take up 80% of the space between data points. The position and width of the bars are controlled by the XmNbarSize and XmNbarPosition resources which are described below, and are pictured in Figure 9-1.

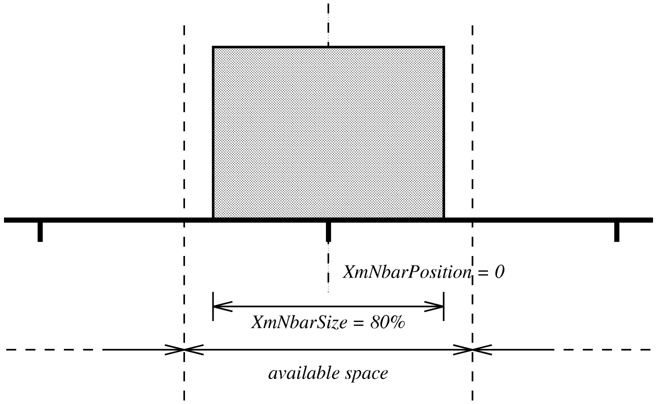


Figure 9-1. The XmNbarSize and XmNbarPosition resources

When you display more than one BarPlot in the same Plotter widget, you can use the XmNbarSize and XmNbarPosition resources to adjust the position and width of each of the bars so that they do not overlap at any data point. Usually, however, there is no need to set the position and size of BarPlot widgets--when multiple BarPlot widgets appear in the same Plotter, they can automatically arrange themselves to share the available space. If two or more BarPlot children of the same Plotter widget have the same number of data points, spaced at the same interval on the X axis, and have the same values for their XmNbarPosition and XmNbarSize resources, (when all have the default values, for example) then they are assumed to be a "group", and those position and size resources apply to the group as a whole. Grouped bars are placed side-by-side within in the available space specified by their XmNbarSize resource; there is no horizontal spacing between them.

The only time you need to use the XmNbarSize and XmNbarPosition resources, then, is when you want to position bars that are not grouped so that they do not overlap one another. Suppose, for example, that you wanted to display six different bar plots in a Plotter widget. If the variables you were plotting broke naturally down into two distinct types, then it might be useful to arrange them in two groups positioned to the left and the right of each X data point, with a small horizontal space between them. You could arrange this by specifying one pair of size and position values for three of the plots and different pair of values for the other three plots. The plots that shared values would automatically arrange themselves, but the two groups would be positioned independently. Figure 9-2 illustrates a case like this, using two groups of two bars. One group of bars is stacked, as explained below.

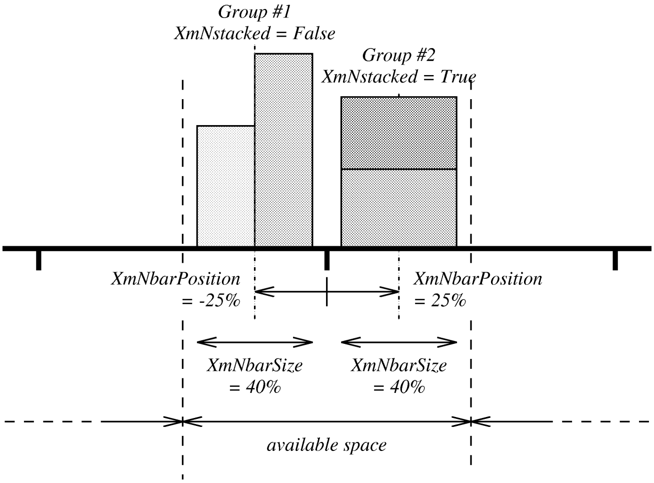


Figure 9-2. Independently positioned groups of bars

When multiple BarPlot widgets in a Plotter have the same number of data points, share the same X values for those points, and have the same values of the XmNbarSize and XmNbarPosition resources, they form a "group", as described above, and are, by default, arranged side by side. If the XmNstacked resource is set to True on each bar in the group, however, the bars will be stacked, as shown in Figure 9-2 to form a bar which has a height that is the sum of the heights of the individual bars. If the bars have different patterns or colors, then this total height will appear as a stack of bars, each making its contribution to the total. This is the simplest way to understand stacked bars--just as a different way of arranging a related group of bars. Note that BarPlot widgets must have matching values for their XmNstacked resource in order to be members of the same group.

The resources that control bar positioning are documented below:

XmNbarSize

The width of a bar, or of a group of related bars (groups of bars are explained in the paragraphs above) as a percentage of the available space. The default value for this integer resource is 80, which specifies that the bar or group of bars should use 80% of the pixels between two adjacent data points on the X axis.

XmNbarPosition

The position of a bar, or group of bars within the space available to it. The available space is the region centered on the X data point and extending half way to the adjacent data points on the left and the right. The default value for this resource is 0 which indicates that the bar, or group of bars, should be centered over the data point. Legal values for this resource are integers between -50 and 50. If you add 50 to the value of this resource, then it becomes the position of the center of the bar or group, expressed as a percentage from 0 to 100 of the available space. Note, however, that because this resource specifies where the center of the bar or group will go, you must also take XmNbarSize into account when setting this value: XmNbarPosition plus or minus half of XmNbarSize must be between -50 and 50.

XmNstacked

A Boolean that specifies whether the bars in a group (groups of bars are explained in the introductory paragraphs of this subsection) will be stacked one on top of another. The default is False, which specifies that grouped bars will be arranged side-by-side.

XmNorigin

The position on the Y axis from which the bars extend up or down. The default of 0.0 is the correct choice for most data, but for some data sets there will be a more natural choice.

9.2.5 Data Resources

The BarPlot widget requires you to specify only the Y values for the bars--it assumes that the bars are evenly spaced along the X axis. There are two resources that specify these X values:

XmNxMin

The X coordinate of the first bar in the plot. The default value is 1.0.

XmNxInterval

The distance between adjacent bars on the X axis. The default value is 1.0.

Note that many bar charts uses textual labels on the X axis, rather than numeric labels. In this case, the values you specify for XmNxMin and XmNxInterval are not themselves important; what is important is only that they match the bounds of the X axis, and that the labels on the Axis are aligned with the bars. Also note that the default value for XmNxMin is 1.0, not 0.0, which is the default for the various XiPlotData constant resources. Because the bars of a BarPlot take up space along the Axis, you should not set your X axis minimum to the same value as XmNxMin, or a portion of the bar width may be truncated (depending on XmNbarPosition and XmNbarSize, of course)

The number of bars in the plot, and the Y value for each bar are specified by a full set of seven XiPlotData resources, listed in Table 9-1. These resources, XmNnumValues, XmNvalues, XmNvalueType, XmNvalueSize, XmNvalueOffset, XmNvalueMultiplier, and XmNvalueConstant, are not described in detail here. Chapter 4, *Plot Data Representation* describes the XiPlotData abstraction and makes the meaning of each of these resources evident.

When BarPlot data is stored in certain commonly used forms, the BarPlot widget provides convenience functions for specifying data which are often easier to use than these resources. The next section describes these functions.

9.3 Specifying BarPlot Data

The most general way to specify data for a BarPlot widget is with the XiPlotData resources described in the previous section. It is often easier, however, to use one of the convenience functions provided by the BarPlot widget for this purpose. The subsections below document these functions.

9.3.1 Arrays of Doubles, Floats, and Ints

The simplest of the BarPlot convenience functions allow you to specify the Y values for the bars as arrays of double, float, or int. They also provide xmin and xinterval arguments that specify the X value for each bar in the same way that the XmNxMin and XmNxInterval resources do. These functions are shown in Figure 9-3.

```
void XiBarPlotSetDoubleValues(Widget w, double xmin, double xinterval,
                              double *values, Cardinal num);
void XiBarPlotSetFloatValues(Widget w, double xmin, double xinterval,
                             float *values, Cardinal num);
void XiBarPlotSetIntValues(Widget w, double xmin, double xinterval,
                           int *values, Cardinal num);
```

`w` The BarPlot widget that is having its data values set.

`xmin` The X coordinate of the first bar in the plot.

`xinterval`

The distance between bars along the X axis.

`values`

An array of Y values for the bars of the plot. This array is of type double, float, or int, depending on the function.

`num`

The number of bars in the plot. There must be at least this many elements in the values array.

Figure 9-3. Simple functions for specifying BarPlot data

```
void XiBarPlotSetValues(Widget w, double xmin, double xinterval,
                        XtPointer vals, XiPlotDataType type,
                        Cardinal size, Cardinal offset, Cardinal num);
```

`w` The BarPlot widget that is having its data set.

`xmin, xinterval`

The X coordinate of the first bar, and the X interval between bars.

`vals, type, size, offset`

An array of structures containing the bar values, the type of each value, the size of each structure, and the offset of the value within the structure. These arguments correspond to the BarPlot XiPlotData resources.

`num`

The number of bars in the plot. The vals array must have at least this many elements in it.

Figure 9-4. XiBarPlotSetValues()

9.3.2 Specifying Data in Structures

Sometimes, you may want to produce a bar chart from values that are stored in structures, rather than in simple arrays, or are of some type other than double, float, or int. In these cases, you can use XiBarPlotSetValues() to specify BarPlot data. This function has arguments that correspond to all of the X and Y data resources, except for XmNvalueConstant and XmNvalueMultiplier. This means it provides all the flexibility of the data resources, except for the linear transform capability. The signature for XiBarPlotSetValues() is shown in Figure 9-4.

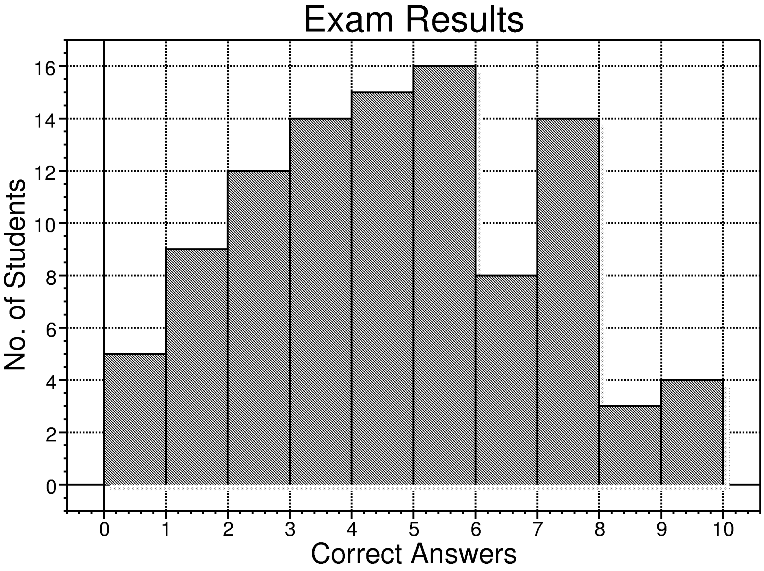
9.3.3 Changing Data in Place

To change the data displayed by a BarPlot widget, you can use one of the functions described above to pass new data to the widget, you can set the data resources to new values, or, if the number of new bars is the same as the old, you can simply overwrite the old data with the new data. If you do this, you must notify the widget that its data has changed by calling XiBarPlotDataChanged().

```
void XiBarPlotDataChanged(Widget w);
```

`w`	The BarPlot widget that has had its data values changed in place.

Calling this function will cause the widget to reread and redisplay its data, extracting it from your application data structures however you previously specified.


Exam Results
No. of Students
16
14
12
10
8
6
4
2
0
0
1
2
3
4
5
6
7
8
9
10
Correct Answers

10

The HistoPlot Widget

The HistoPlot widget is a subclass of BarPlot that displays histograms. A regular bar plot simply plots the data points you specify. A histogram is different: instead of displaying the data itself, it displays the distribution of the data. In a histogram, the X axis is divided into intervals and a bar is drawn for each of those intervals. The bar displays the number of data points that fall within the interval. (These intervals are often called "bins".) Histograms are useful, for example, when studying test scores, the size of network packets, or performing any other statistical analysis on collected data.

HistoPlot Synopsis

Class Name: XiHistoPlot

Class Hierarchy: RectObj → XiPlot → XiHistoPlot

Header File: <*Xi/HistoPlot.h*>

Class Pointer: `xiHistoPlotWidgetClass`

Constructor:

```
XiCreateHistoPlot (Widget parent,String name,
                   ArgList args,
                   Cardinal num_args);
```

Because the HistoPlot widget is a subclass of BarPlot, it inherits all the resources of that widget class that control the size, position, color, fill pattern, and so on for the bars. New resources let you specify the size and position of the "bins". Other resources let you specify the raw data to be analyzed--the HistoPlot widget will automatically count the number of these raw data values that fall into each bin.

This chapter explains how to create a HistoPlot widget, how to specify raw data for the histogram, and the resources that control the histogram.

10.1 Creating a HistoPlot

You can create a HistoPlot with the function XiCreateHistoPlot(), which is a standard Motif-style widget constructor, or with XtCreateWidget() or XtVaCreateWidget() with the widget class pointer xiHistoPlotWidgetClass. Both the constructor function and the widget class pointer are declared in <*Xi/HistoPlot.h*>.

Example 10-1 shows how you might create a HistoPlot widget:

Example 10-1. Creating a HistoPlot widget

```
#include <Xi/Plotter.h>    /* always include this */
#include <Xi/HistoPlot.h>  /* declares constructor and class pointer
*/

Widget plotter;  /* assume this is already created */
Widget histogram;
static double min = 1.0;
static double max = 12.0;
static double size = 1.0;

histogram = XtVaCreateManagedWidget("histogram", xiHistoPlotWidget-
Class,
                                    plotter,
                                    XmNmin, &min,
                                    XmNmax, &max,
                                    XmNbinSize, &size,
                                    NULL);
```

10.2 HistoPlot Resources

The HistoPlot widget inherits all the resources of the BarPlot widget. It defines five new resources that specify how the data are to be counted into bins, and a set of XiPlotData resources for specifying the data. The HistoPlot widget also overrides the default value of the BarPlot XmNbarSize resource, setting it to 100--this means that, by default, the bars in a histogram will touch one another.

The new HistoPlot resources are summarized in Table 10-1 and are documented below. Note that a number of these new resources are of type double, and special techniques are required to set them. (See Chapter 3, *Plotter Data Types and Resources* .) Chapter 9, *The BarPlot Widget* documents the resources of the BarPlot widget. You will need to use these resources to specify such things as bar color, fill pattern, spacing, and so on. For convenience, tables of BarPlot and HistoPlot resources appear in the quick reference section at the end of this manual.

Table 10-1. HistoPlot Resources

Name	Class	Type	Default	CSG	Pg
Bounds					
XmNmin	XmCMin	Double	0.0	CSG	149
XmNmax	XmCMax	Double	100.0	CSG	149
XmNcomputeBounds	XmCComputeBounds	Boolean	FALSE	CSG	149
Bins					
XmNnumBins	XmCNumBins	Short	-1[a]	CSG	150
XmNbinSize	XmCBinSize	Double	10.0	CSG	150
XmNbarSize[b]	XmCBarSize	Short	100	CSG	
Data[c]					
XmNnumRawValues	XmCNumValues	Cardinal	0	CSG	150
XmNrawValues	XmCValues	Pointer	NULL	CSG	150
XmNrawValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	150
XmNrawValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	150
XmNrawValueOffset	XmCValueOffset	Cardinal	0	CSG	150
XmNrawValueConstant	XmCValueConstant	Double	0	CSG	150
XmNrawValueMultiplier	XmCValueMultiplier	Double	1.0	CSG	150

[a] *An unspecified value; will be computed based on XmNbinSize.*

[b] *This is a BarPlot resource; HistoPlot changes the default value.*

[c] *Do not set these resources when using a DBPak subclass of this widget.*

10.2.1 Boundary and Bin Resources

The HistoPlot widget has five resources that specify how the data values are to be counted into their "bins":

XmNmin

A double resource that specifies the lower boundary of the first bin in the plot. Any data values that are less than XmNmin will not be counted in any of the bins. The default is 0.0.

XmNmax

A double resource that specifies the upper boundary of the last bin in the plot. Any data values that are greater than XmNmax will not be counted in any of the bins. The default is 100.0.

XmNcomputeBounds

A Boolean that specifies whether the HistoPlot widget should automatically compute values for XmNmin and XmNmax based on the minimum and maximum specified data values. The default is False. This resource can be useful if you do not have any idea about the distribution of the data to begin with, and when there are not natural lower and upper bounds that can be placed on the data. Setting this resource to True gives you a kind of autoscaling of your data so that it takes up all the available horizontal space in the Plotter widget, but is not always appropriate. When displaying a distribution of test scores percentages, for example, you probably want the lower bound to be 0.0, even if the minimum score on the test was 40.

XmNnumBins

The number of equal bins that the interval between XmNmin and XmNmax should be divided into. The default value of this resource is -1, a special value that indicates that the number of bins should instead be computed from the XmNbinSize resource. If you do set this resource to something other than the default, then the XmNbinSize resource will be ignored.

XmNbinSize

A double resource that specifies how large each bin should be. The default value of this resource is 10.0, but it is only used if the XmNnumBins resource is not set. If that resource is set, then XmNbinSize is computed from it. If you use XmNbinSize, then you should be sure that the interval between XmNmin and XmNmax is evenly divisible by XmNbinSize; if it is not, the last bin in the plot will be "narrower" than the others (though the bar that represents it on the screen will be as wide as all the others, of course).

10.2.2 Data Resources

The HistoPlot has a standard complement of XiPlotData resources for specifying the raw data that is to be counted into the bins of the histogram: XmNnumRawValues, XmNrawValues, XmNrawValueType, XmNrawValueSize, XmNrawValueOffset, XmNrawValueConstant, and XmNrawValueMultiplier. You can specify data values with these resources as explained in Chapter 4, *Plot Data Representation*. The HistoPlot widget also provides convenience functions for setting data values that may often be more convenient than using these values. These functions are documented in the section below.

These HistoPlot data resources have the "rawValue" prefix to distinguish them from the "value" resources of the BarPlot superclass. When using the HistoPlot widget, you should never set the data resources of the BarPlot widget; the HistoPlot widget will analyze the "raw values" into bins, and then set the BarPlot data resources itself to display the contents of each bin.

10.3 Specifying HistoPlot Data

The most general way to specify raw data to be sorted and displayed by a HistoPlot widget is with the XiPlotData resources described above. The HistoPlot widget also provides convenience routines, that are often easier to use than those resources. They are described in the subsections below.

10.3.1 Arrays of Doubles, Floats, and Ints

The simplest of the HistoPlot convenience functions allow you to specify raw data in a single array of double, float, or int values. These functions are shown in Figure 10-1.

```
void XiHistoPlotSetDoubleValues(Widget w, double *values, Cardinal
       num);
void XiHistoPlotSetFloatValues(Widget w, float *values, Cardinal num);
void XiHistoPlotSetIntValues(Widget w, int *values, Cardinal num);
```

`w` The HistoPlot widget that is having its raw data specified.

`values` An array of raw data to be displayed as a histogram by the widget. The elements of the array are of type double, float, or int, depending on the function.

`num` The number of values to be included in this histogram analysis. There must be at least this many elements in the values array. Note that this is not the number of bars that will appear in the plot.

Figure 10-1. Specifying HistoPlot data in simple arrays

Note that these functions specify only the raw data for the HistoPlot. You must still specify the bounds for the histogram, and the number of bins or the bin size. You do not have to specify any X or Y data for this BarPlot superclass, however; the HistoPlot uses the raw data, and the boundary and bin specifications to set the BarPlot resources appropriately.

10.3.2 Specifying Data from Structures

Sometimes the raw data for a histogram may be stored as a field within a larger structure, or may be stored as some type other than double, float, or int. In this case, you can specify that data with the function XiHistoPlotSetValues(). This function has resources that correspond to each of the raw data resources of the HistoPlot except for XmNrawDataConstant and XmNrawDataMultiplier. This means that this convenience function has all the flexibility of the data resources, except for the linear transform capability they allow. The signature of XiHistoPlotSetValues() is shown in Figure 10-2.

```
void XiHistoPlotSetValues(Widget w,
                           XtPointer vals, XiPlotDataType type,
                         Cardinal size, Cardinal offset, Cardinal num);
```

`w` The HistoPlot widget that is having its raw data set.

`vals, type, size, offset`

An array of structures containing the raw data values, the type of each value, the size of each structure, and the offset of the value within the structure. These arguments correspond to the HistoPlot XiPlotData resources.

`num` The number of elements in the vals array that are to be used in the histogram. vals must have at least this many elements.

Figure 10-2. XiHistoPlotSetValues()

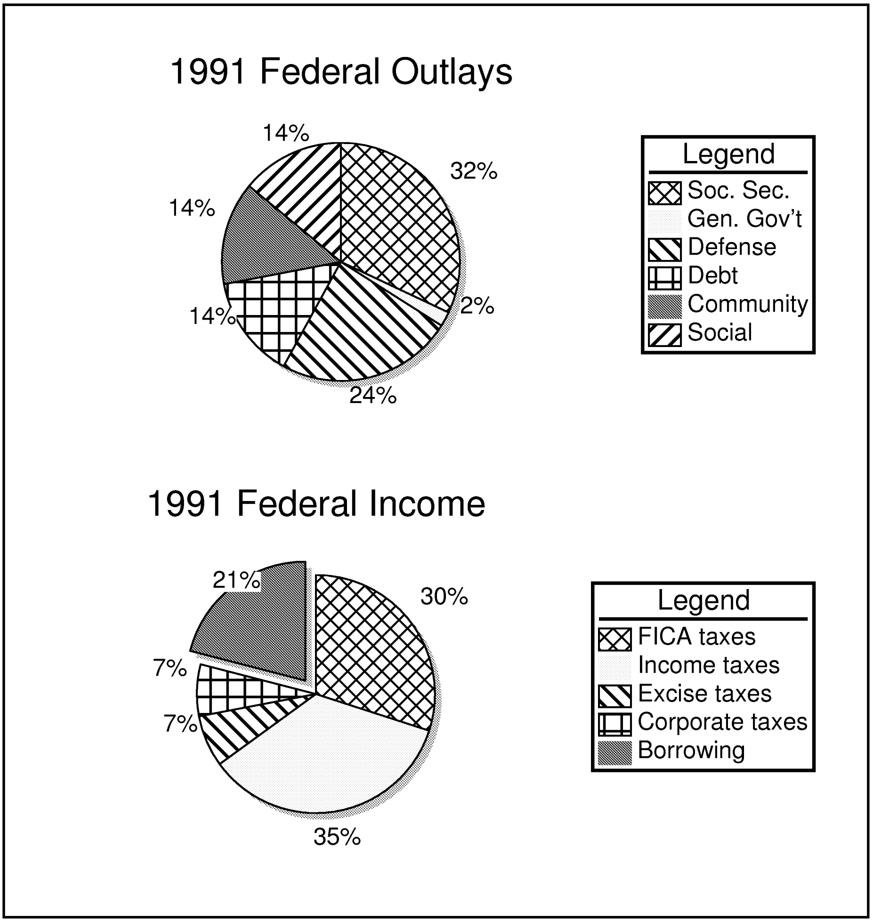
10.3.3 Changing Data in Place

To change the raw data used by a HistoPlot widget, you can use one of the functions described above to pass new raw data values to the widget, or you can set the XiPlotData resources to new values, or, if the number of raw data values remains the same, you can simply overwrite the old data with the new data. If you do this, you must notify the widget that its data has changed by calling XiHistoPlotDataChanged().

```
void XiHistoPlotDataChanged(Widget w);
```

`w` The HistoPlot widget that has had its data values changed in place.

Calling this function will cause the widget to re-read, re-analyze, re-display its data, extracting it from your application data structures however you previously specified.


1991 Federal Outlays
14%
32%
14%
2%
14%
24%
Legend
Soc. Sec.
Gen. Gov't
Defense
Debt
Community
Social
1991 Federal Income
21%
30%
7%
7%
35%
Legend
FICA taxes
Income taxes
Excise taxes
Corporate taxes
Borrowing

11

The PiePlot Widget

The PiePlot widget is used to display "slices" in a pie chart. This kind of graph is useful to display the proportional contribution of a number of quantities that sum to some total value. It is useful to display business income and expenditures by category, for example. Human perception seems to be good at understanding angles, and so this angular representation of values is a good way to display the relative size of values. Because pie charts are weighted (there are always 360 degrees in a pie, no matter what the quantities being displayed are) they can also be useful when the values being displayed are too large or too small for a useful intuitive understanding.

PiePlotPlot Synopsis

Class Name: XiPiePlot

Class Hierarchy: RectObj → XiPlot → XiPiePlot

Header File: <*Xi/PiePlot.h*>

Class Pointer: `xiPiePlotWidgetClass`

Constructor:

```
XiCreatePiePlot (Widget parent, String name,
                 ArgList args,
                 Cardinal num_args);
```

The PiePlot widget is a little different from other Plot widgets--while the LinePlot and the BarPlot widgets display complete line plots and bar charts, each PiePlot widget displays only a single wedge of a pie chart. What this means is that each wedge of a pie chart can have its own fill pattern, color, and, importantly, its own entry in the Plotter legend. Rather than supplying an array of data values for the pie chart as a whole, each PiePlot widget has a single XmNvalue resource. The values for each PiePlot in the Plotter are added together, and the size of each pie slice is determined by the contribution of each value to the whole. While this use of multiple PiePlot widgets to create a single pie chart may seem strange at first, it actually makes it simpler to create pie charts, because there is no need for awkward resources like arrays of strings or arrays of fill patterns.

As mentioned above, each PiePlot wedge can have its own color and/or fill pattern. Additionally, each wedge will be outlined with a line of the color and width you specify. Wedges can have shadows, much like bar plot and line plot shadows, and individual wedges can be "exploded" slightly away from the center of the pie for emphasis.

As with all Plot types, each PiePlot can have an entry in the legend with any appropriate string. PiePlots can also have a string placed near their wedge. Resources control the font or fonts, color and position of this label, and the label string may optionally contain a printf()-style substitution which will be replaced with the percentage the wedge contributes to the whole, or the absolute value represented by the wedge.

Although it is common to see "3D" pie charts that have been rotated so that the circular pie becomes elliptical, the PiePlot widget cannot display this kind of chart--rotating the pie in this way distorts the data it is supposed to represent, and so is not a valid display mode. The PiePlot shadows do provide a three dimensional appearance and can add impact to a pie chart while still keeping it circular and undistorted.

The following sections show how to create PiePlot widgets, and explain each of the PiePlot resources.

11.1 Creating PiePlot Widgets

You can create a PiePlot widget with the function XiCreatePiePlot(), which is standard Motif-style widget constructor, or you can use XtCreateWidget() or XtVaCreateWidget() with the widget class pointer xiPiePlotWidgetClass. Both the constructor function and the widget class pointer are declared in the header <*Xi/PiePlot.h*>.

Example 11-1 demonstrates how you might create PiePlot widgets. Since you always need more than one PiePlot widget to make an interesting piechart, creating the widgets in a loop, as shown in this example, is often a useful technique.

The example points out another difference between the PiePlot widget and other Plot types: the X and Y axes are not meaningful for piecharts. When you are displaying PiePlot widgets the easiest thing to do, as in the example, is simply to unmanage the X and Y axes. This will make them disappear. Sometimes, when displaying presentation graphics, you may want to keep the axes and their grid, simply because they add interest to the display. When you do this, you will probably want to get rid of the Axis titles, labels, tic marks, and origin by setting the Axis resources XmNtitle, XmNdrawLabels, XmNmarkLength, XmNsubmarkLength, and XmNdrawOrigin. You might also want to use XiAxisSetMarks() to explicitly control the distance between the lines of the grid and subgrid.

Example 11-1. Creating a piechart with PiePot

```
#include <Xi/Plotter.h>  /* always include this */
#include <Xi/PiePlot.h>  /* declares constructor and class pointer */

Widget plotter;  /* assume this is already created */
Widget wedges[5];
char name[10];
int i;

/* data is federal government income for fiscal 1991, in billions of $*/
static double values[] = {316, 369, 74, 74, 221};
static String legends[] = {
    "FICA taxes, etc.",
    "Personal income taxes",
    "Excise taxes, etc.",
    "Corporate income taxes",
    "Borrowing"
};
static XiFillPattern patterns[] = {
    XiFillPatternGray1, XiFillPatternGray2,
    XiFillPatternGray3, XiFillPatternGray4,
    XiFillPatternGray5,
};

for(i=0; i < XtNumber(wedges); i++) {
    sprintf(name, "wedge%d", i);
    wedges[i] = XtVaCreateWidget(name, xiPiePlotWidgetClass, plotter,
                                 XmNvalue, &values[i], /* a double
                                 resource */
                                 XmNlegendName, legends[i],
                                 XmNfillPattern, patterns[i],
                                 XtVaTypedArg, XmNlineColor, XmRString,
                                      "maroon", 7,
                                 XmNlabel, "%.0g%%",
                                 NULL);
}

/* Plots all have to be managed, or they won't appear */
XtManageChildren(wedges, XtNumber(wedges));

/* Pie charts don't need axes, so we hide them by unmanaging. */
XtUnmanageChild(XiPlotterXAxis(plotter));
XtUnmanageChild(XiPlotterYAxis(plotter));
```

Table 11-1. PiePlot Resources

Name	Class	Type	Default	CSG	Pg
Fill Pattern					
XmNfillColor	XmCForeground	Pixel	XmNforeground[a]	CSG	159
XmNfillBackground	XmCBackground	Pixel	XmNbackground[b]	CSG	159
XmNfillPattern	XmCFillPattern	FillPattern	XiFillPatternSolid	CSG	159
XmNfillOpaque	XmCFillOpaque	Boolean	TRUE	CSG	159
Lines					
XmNlineColor	XmCForeground	Pixel	XmNforeground[a]	CSG	160
XmNlineWidth	XmCLineWidth	Dimension	0	CSG	160
XmNdrawRadius	XmCDrawRadius	Boolean	TRUE	CSG	160
Labels					
XmNlabelFontList	XmCFontList	XmFontList	XmNfontList[c]	CSG	161
XmNlabelPSFontListf	XmCPSFontList	XiPSFontList	XmNpsFontList[d]	CSG	161
XmNlabelString	XmCLabelString	XmString	NULL[e]	CSG	162
XmNlabel	XmCLabel	String	NULL	CSG	162
XmNlabelSize	XmCFontSize	Dimension	12	CSG	162
XmNlabelStyle	XmCFontStyle	String	NULL	CSG	162
XmNlabelColor	XmCForeground	Pixel	XmNforeground[a]	CSG	163
XmNlabelRadius	XmCLabelRadius	Dimension	38	CSG	163
XmNshowValue	XmCShowValue	Boolean	FALSE	CSG	163
Shadows					
XmNdrawShadow	XmCDrawShadow	Boolean	FALSE	CSG	160
XmNshadowColor	XmCForeground	Pixel	XmNforeground[a]	CSG	160
XmNshadowPattern	XmCShadowPattern	FillPattern	XiFillPatternSolid	CSG	161
XmNshadowOpaque	XmCShadowOpaque	Boolean	TRUE	CSG	161
XmNshadowXOffset	XmCShadowOffset	Position	4	CSG	161
XmNshadowYOffset	XmCShadowOffset	Position	4	CSG	161
Layout					
XmNxPosition	XmCXPosition	Dimension	50	CG	164
XmNyPosition	XmCYPosition	Dimension	50	CG	164
XmNradius	XmCRadius	Dimension	35	CG	164
XmNexplodeRadius	XmCExplodeRadius	Dimension	0	CSG	165
XmNstartAngle[f]	XmCStartAngle	Position	90	CG	165
Data					
XmNvalue[g]	XmCValue	Double	0.0	CSG	163

[a] *Inherited from the Plotter XmNforeground resource.*

[b] *Inherited from the Plotter XmNplotAreaColor resource.*

[c] *Inherited from the Plotter XmNfontList resource.*

[d] *Inherited from the Plotter XmNpsFontList*

[ed] *When an XmString resource is unset, its value is created from the corresponding String resource, using a font tag derived from the corresponding style and size resources.*

[f] *This resource is ignored except for the first wedge in a pie chart.*

[g] *Do not set this resource when using a DBPak subclass of this widget.*

11.2 PiePlot Resources

The PiePlot widget has resources that control the color and fill pattern of the wedge, the way the wedge is outlined, shadowed, and labeled. Unlike other Plot types, however, the PiePlot widget only has one resource for the data it is to display, rather than one or more sets of XiPlotData resources. This is because a PiePlot widget only displays a single slice of a pie chart, and there is no need for an array of data values for each widget.

The PiePlot widget resources are summarized in Table 11-1, and are explained in the following subsections.

11.2.1 PiePlot Fill Pattern

Like the BarPlot widget, the PiePlot widget fills its wedge with a specified color or pattern. These resources specify how the wedge is filled:

XmNfillColor

The solid color of the wedge, or the foreground color of the fill pattern for the wedge. The default value for this resource is inherited from the XmNforeground resource of the parent Plotter widget.

XmNfillBackground

The background color of the pattern used to fill the PiePlot wedge. If XmNfillPattern is XiFillPatternSolid, or XmNfillOpaque is False, then this resource is not used. If you do not set this resource, its value is inherited from the XmNplotAreaColor resource of the parent Plotter widget. This default value will make patterned wedges appear transparent.

XmNfillPattern

The pattern to be used to fill the PiePlot wedge. The default is the special value XiFillPatternSolid which specifies that the wedge should be filled with a solid color rather than a pattern. Chapter 3, *Plotter Data Types and Resources* explains the XiFillPattern abstraction, and that chapter and the quick reference section at the end of this manual contain lists of the predefined fill patterns. You can also define custom fill patterns of your own; Chapter 18, *Defining Custom FIll Patterns* explains how to do this.

XmNfillOpaque

A Boolean that specifies whether pattern fills should be "opaque" or "transparent". With an opaque fill, the PiePlot widget draws both the foreground and background bits of the pattern, so that anything under the bars is erased. This is the default, and is almost always the appropriate setting for this resource. If you set it to False, then only the foreground bits of the fill pattern will be drawn, and anything previously drawn underneath the bars (such as grid lines) will show through.

11.2.2 PiePlot Outline

The PiePlot widget always draws an outline around the outside of each wedge, and optionally outlines the sides of each wedge. These resources specify how the outlining is done:

XmNlineColor

The color of the line that outlines the PiePlot wedge. If this resource is not set, it inherits the value of the XmNforeground resource of the parent Plotter widget.

XmNlineWidth

The width, in pixels, of the outline drawn around each wedge. The default is 0, which is a special value that specifies efficiently-drawn lines one pixel wide.

XmNdrawRadius

A Boolean resource that specifies whether the PiePlot widget should draw the radius lines from the center of the pie to its outside arc, or whether it should limit its outlining to the arc alone. The default is True which specifies that the radius lines should be drawn as part of the outline.

11.2.3 PiePlot Shadows

Each PiePlot wedge can be drawn with a shadow underneath it. This shadow gives a 3D effect to the pie chart and adds impact to presentation graphics, without distorting the pie into an ellipse. The following resources specify the attributes of the wedge shadow.

XmNdrawShadow

A Boolean resource that specifies whether a shadow should be drawn under the BarPlot wedge. The default is False.

XmNshadowColor

The color of the shadow to draw under the wedge. If you do not set this resource, its value is inherited from the XmNforeground resource of the parent Plotter widget. Note that when this shadow color is combined with the stipple pattern specified by XmNshadowPattern, the apparent shadow color that results can be significantly lighter than the shadow color alone would be.

XmNshadowPattern

The fill pattern to use as a stipple for the shadow. The default is XiFillPatternGray3, which is a 50% gray pattern. The gray fill patterns all make good shadow stipples, but many of the other pre-defined patterns are not appropriate for this use.

XmNshadowOpaque

A Boolean resource that specifies whether the shadow drawn under the PiePlot wedge should be opaque or not. The default is True, which specifies that both the foreground and the background bits of the shadow pattern will be drawn. The background bits will be given the same color as the XmNplotAreaColor of the Plotter, effectively erasing anything under the shadow. If you set this resource to False, then only the foreground bits of the pattern will be drawn, and any anything "under" the shadow (such as grid lines) will show through where the pattern is not drawn.

XmNshadowXOffset, XmNshadowYOffset

The X and Y offsets, in pixels, of the BarPlot wedge's shadow from the wedge itself. The default for each of these resources is 2 pixels. Larger values increase the separation between wedge and shadow, making the wedge appear three-dimensionally further above the plotter background. Negative values are legal for both of these resources; varying their values moves the position of the imaginary light source that casts the shadows.

11.2.4 PiePlot Labels

Each PiePlot wedge can have a label associated with it. The following resources control the font, contents, position, and color of the label:

XmNlabelFontList

The font or fonts in which the PiePlot label should be displayed. If this resource is unspecified, its value is inherited from the parent Plotter's XmNfontList resource.

XmNlabelPSFontList

The list of PostScript fonts to be used to display the PiePlot's label in hardcopy. If no value is specified for this resource, its default value is inherited from the parent Plotter XmNpsFontList resource. You need only specify this resource if XmNlabelFontList contains fonts outside of the "standard" font families--Adobe Times, Helvetica, Courier, New Century Schoolbook, and Symbol. See Section 3.3, *Fonts and Displayed Text* for information on specifying PostScript font lists.

XmNlabelString

The XmString label to be displayed by the PiePlot. If no XmString is specified for this resource, then one will be created using the XmNlabel string, and a font list tag constructed from the XmNlabelStyle and XmNlabelSize resources, as described in Section 3.3, *Fonts and Displayed Text*. By using this XmNlabelString resource to specify an XmString directly, you get the flexibility to display labels with multiple fonts. What you give up, however is the formatting of the XmNlabel resource which allows you to automatically display the value or percentage of any PiePlot wedge in the label. Setting XmNlabelString will override any value previously created from the XmNlabel string. If you query this resource, the Plotter returns its private, internal copy of the XmString, which you must not modify or free.

XmNlabel

The label to be drawn for the wedge, if no XmNlabelString is specified. Note that this resource is an ordinary null-terminated character string, not a Motif XmString. The default value is NULL which specifies that the wedge will be unlabeled. The specified string may contain a single printf() substitution for a double value. (i.e., a %g, %f or %e substitution, with any legal modifiers.) Depending on the XmNshowValues resource, the PiePlot widget will replace this substitution string with either the absolute value of the wedge's data, or the percentage that the wedge comprises of the whole.

Once the value or percentage has been substituted into the XmNlabel string, the resulting formatted string is converted to an XmString using a font list tag constructed from the XmNlabelStyle and XmNlabelSize resources, and this newly created XmString becomes the new value of the XmNlabelString resource. See Section 3.3 for details.

Note that XmNlabel is not the same as the Plotter constraint resource XmNlegendName--XmNlabel specifies a string to be displayed within the plotting area, near an individual wedge, while XmNlegendName specifies a label to be displayed in the plotter legend.

XmNlabelSize

The point size of the PiePlot label. This value is used in the font list tag of the XmNlabel resource, if no XmNlabelString is specified. The default is 12.

XmNlabelStyle

The typeface of the PiePlot label. This value is used in the font list tag of the XmNlabel resource, if no XmNlabelString is specified. The default is NULL. When you set this resource, the string value is copied, and when you query it, you must not modify or free the returned value.

XmNlabelColor

The color in which the PiePlot label is to be drawn. If this resource is not set, its value is inherited from the XmNforeground resource of the parent Plotter widget.

XmNlabelRadius

The distance between the center of the pie and the closest edge of the label, expressed as a percentage of the width or height of the Plotter plotting area. The default value is 38. If this resource is greater than XmNradius, then the labels will appear outside of the wedges. If it is smaller than XmNradius, then the labels will appear at least partially inside the wedges. See the XmNradius resource below for more explanation of how the value of this resource is interpreted.

XmNshowValue

A Boolean resource that specifies whether the PiePlot should substitute its percentage, or its absolute value into the XmNlabel string. The default value is False, which specifies that the PiePlot should substitute its percentage. If you set XmNshowValue to True, the PiePlot widget will show its value in the label, rather than its percentage.

11.2.5 PiePlot Data

Unlike all other Plot types, the PiePlot does not have any XiPlotData resources--each wedge requires only a single data value, so no arrays of values are required. That value is specified with a single resource:

XmNvalue

The absolute value that this slice of pie represents. This is "raw data," not a percentage; this value is added to the values of all the other PiePlot wedges in the Plotter, and its portion of the total is used to determine the total size of the wedge. Note that this is a double resource which requires some special care to set; see Chapter 3, *Plotter Data Types and Resources*.

11.2.6 PiePlot Positioning

The PiePlot widget has resources that control the size and center position of the pie. Rather than using absolute pixel positions and distances, these resources are percentages of the space available, so that a piechart will grow or shrink when the Plotter widget grows and shrinks. The resources that control horizontal position are interpreted as percentages of the plotting area width, and resources that control vertical position are interpreted as percentages of the plotting area height. A number of resources specify distance rather than position, however, and these distances are radii, which are not necessarily horizontal or vertical. Since the PiePlot widget does not allow elliptical pie charts, these distance resources are interpreted as a percentage of the smaller of the plotting area width and height.

When displaying pie charts, you may want to make the Plotter widget a fair bit wider than it is high, so that there is room for the legend, and the remaining plotting area is still wider than high. The size of the pie chart will be measured in terms of the smaller height, and there will be extra space available on the left and right for the PiePlot labels, which are always drawn horizontally.

The resources that control PiePlot positioning are the following. PiePlot label positioning is controlled with the XmNlabelRadius resource described above.

XmNxPosition

The horizontal position of the center of the pie, expressed as a percentage of the width of the Plotter plotting area. The default value is 50, which centers the piechart horizontally. All PiePlot widgets that are to form part of the same pie chart must have the same value for this resource. You can create two pie charts in the same Plotter widget by creating two groups of PiePlot widgets which have different center points.

XmNyPosition

The vertical position of the center of the pie, expressed as a percentage of the height of the Plotter plotting area. The default value is 50, which centers the piechart vertically. All PiePlot widgets that are part of the same pie chart must have the same value for this resource.

XmNradius

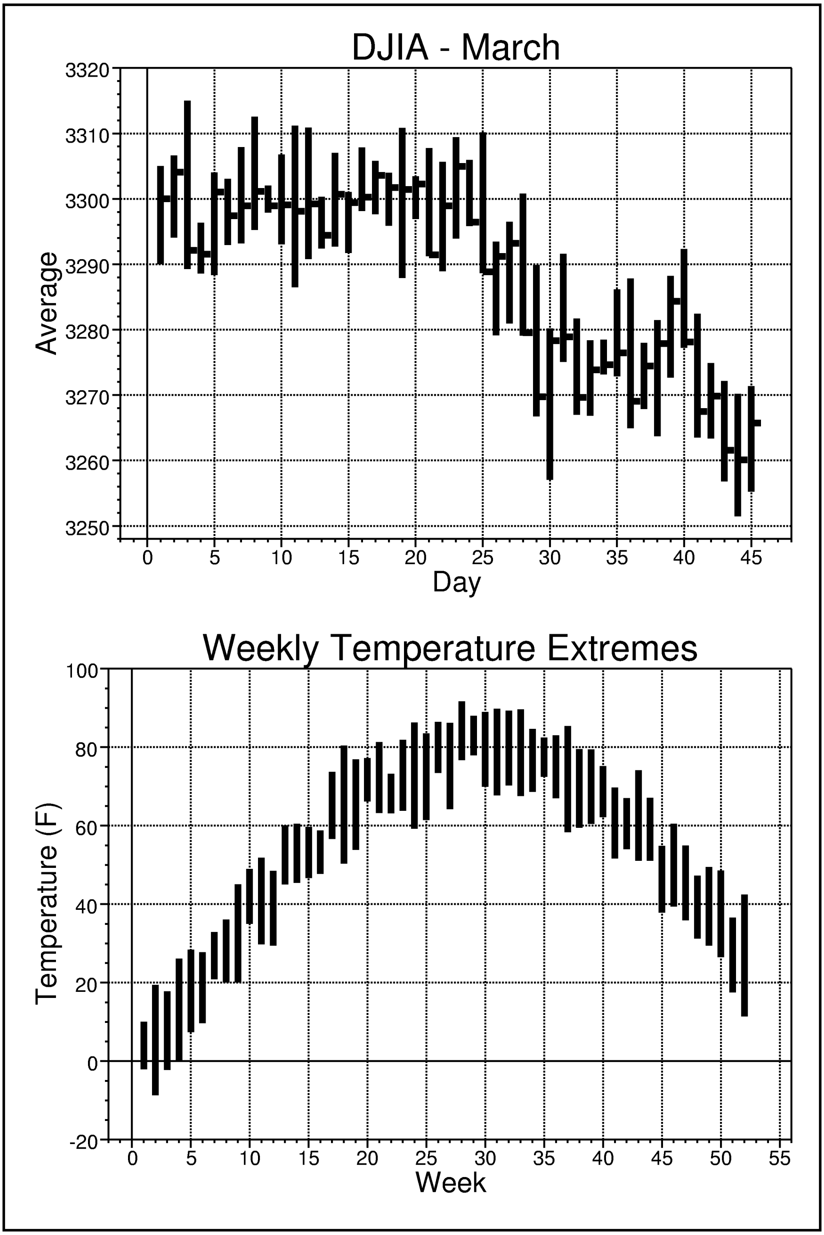
The radius of the PiePlot wedge. This value is specified as a percentage of the smaller of the width and height of the Plotter plotting area. The maximum reasonable value for this resource depends on the center position of the pie, but must obviously never exceed 50, or the piechart will not fit within the plotting area. The default value of 35 leaves room for labels on all sides of the pie. All PiePlot widgets that are part of the same pie chart must have the same value for this resource.

XmNexplodeRadius

The distance by which an individual wedge within a pie chart should be "exploded" away from the center of the pie. This distance is measured as a percentage of the smaller of the width and height of the plotting area, just as the XmNradius resource is measured. The default is 0, which specifies that the wedge should not be exploded at all. Reasonable values are between 5 and 15--this value measures the distance between the center of the pie and the vertex of the wedge, not the "new radius" between the center of the pie and the outside arc of the wedge.

XmNstartAngle

The angle at which the PiePlot wedge should begin. This resource may only be set for the first wedge in a pie chart, and the results will be undefined if it is set for any other wedges. (The first wedge is the first one created, or the wedge with the lowest value for its XmNdrawingOrder constraint resource.) This resource may also only be set when the widget is created, and may not be set at any time after that. Angles are measured counter-clockwise from the positive X axis, and the units are 64ths of degrees (this is the standard for Xlib). The default is 5760 (90\(de \(mu 64), which specifies the 12 o'clock position. PiePlot wedges are arranged clockwise around the circle beginning at this starting angle.


DJIA - March
Average
3320
3310
3300
3290
3280
3270
3260
3250
0
5
10
15
20
25
30
35
40
45
Day
Weekly Temperature Extremes
Temperature (F)
100
80
60
40
20
0
-20
0
5
10
15
20
25
30
35
40
45
50
55
Week

12

The HighLowPlot Widget

The HighLowPlot widget displays graphs of high-low-close data. This widget is typically used to display financial market data, such as the performance of a particular stock or of a market average such as the Dow Jones Industrial Average. With the close data omitted, the widget is also useful for displaying data such as the daily high and low temperatures over some time period.

HighLowPlot Synopsis

Class Name: XiHighLowPlot

Class Hierarchy: RectObj → XiPlot → XiHighLowPlot

Header File: <*Xi/HighLowPlot.h*>

Class Pointer: `xiHighLowPlotWidgetClass`

Constructor:

```
XiCreateHighLowPlot (Widget parent, String
                     name, ArgList args,
                     Cardinal num_args);
```

The high and low values for a data point are represented with a vertical bar drawn between them. The close value for the data point is represented by a horizontal line extending a short distance to the right of the bar. This is a standard display format for market data, used, for example, by *The Wall Street Journal*. HighLowPlot resources allow you to specify the color of the bar and line, the pixel width of the bar, the pixel width of the line, and the horizontal length of the line.

As with all Plot widgets, the HighLowPlot will extract its high, low, and close values from application data structures. You can set resources or use convenience functions to specify how the data is to be extracted. If the X values for the data are evenly spaced, you need only specify the starting point and the X interval between points. You may also specify a complete set of X values, however, which means that you can leave gaps in the data when, for example, there are market holidays and no trading takes place.

This chapter explains how to create a HighLowPlot, documents the resources that control the appearance of the HighLowPlot, and explains the resources and convenience functions that you can use to specify high, low, and close data values for a plot.

12.1 Creating a HighLowPlot

You can create a HighLowPlot widget with the function XiCreateHighLowPlot(), which is a standard Motif-style widget constructor, or you can use XtCreateWidget() or XtVaCreateWidget() with the widget class pointer xiHighLowPlotWidgetClass. Both the constructor function and the widget class pointer are declared in the header file <*Xi/HighLowPlot.h*>.

Example 12-1 demonstrates how you might create a HighLowPlot.

Example 12-1. Creating a HighLowPlot widget

```
#include <Xi/Plotter.h>      /* always include this */
#include <Xi/HighLowPlot.h>  /* declares constructor and class
pointer */

Widget plotter; /* assume this is already created */
Widget plot;     /* the plot we are creating */

/*
 * This plot won't have any meaningful close data, so we set the
 * line length to 0 so close values aren't displayed.
 */
plot = XtVaCreateWidget("temperature", xiHighLowPlotWidgetClass,
plotter,
                        XmNlegendName, "Daily Temperature Extremes",
                        XmNbarWidth, 4,
                        XmNlineLength, 0,
                        NULL);
XtManageChild(plot); /* Don't forget this! */
```

Table 12-1. HighLowPlot Resources

Name	Class	Type	Default	CSG	Pg
Bars					
XmNbarColor	XmCForeground	Pixel	XmNforeground[a]	CSG	171
XmNbarWidth	XmCLineWidth	Dimension	5	CSG	171
XmNlineColor	XmCForeground	Pixel	XmNforeground[a]	CSG	171
XmNlineWidth	XmCLineWidth	Dimension	3	CSG	171
XmNlineLength	XmCLineLength	Dimension	6	CSG	171
Data[b]					
XmNnumValues	XmCNumValues	Cardinal	0	CSG	171
XmNxValues	XmCXValues	Pointer	NULL	CSG	171
XmNxValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	171
XmNxValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	171
XmNxValueOffset	XmCValueOffset	Cardinal	0	CSG	171
XmNxValueConstant	XmCValueConstant	Double	0.0	CSG	171
XmNxValueMultiplier	XmCValueMultiplier	Double	1.0	CSG	171
XmNhighValues	XmCValues	Pointer	NULL	CSG	171
XmNhighValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	171
XmNhighValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	171
XmNhighValueOffset	XmCValueOffset	Cardinal	0	CSG	171
XmNhighValueConstant	XmCValueConstant	Double	0.0	CSG	171
XmNhighValueMultiplier	XmCValueMultiplier		Double	CSG	171
XmNlowValues	XmCValues	Pointer	NULL	CSG	171
XmNlowValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	171
XmNlowValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	171
XmNlowValueOffset	XmCValueOffset	Cardinal	0	CSG	171
XmNlowValueConstant	XmCValueConstant	Double	0.0	CSG	171
XmNlowValueMultiplier	XmCValueMultiplier	Double	1.0	CSG	171
XmNcloseValues	XmCValues	Pointer	NULL	CSG	171
XmNcloseValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	171
XmNcloseValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	171
XmNcloseValueOffset	XmCValueOffset	Cardinal	0	CSG	171
XmNcloseValueConstant	XmCValueConstant	Double	0.0	CSG	171
XmNcloseValueMultiplier	XmCValueMultiplier	Double	1.0	CSG	171

[a] *Inherited from the Plotter XmNforeground resource.*

[b] *Do not set these resources when using a DBPak subclass of this widget.*

12.2 HighLowPlot Resources

The HighLowPlot widget has resources that control the size and color of the "bars" and "lines" that it draws. (The "bar" is the vertical portion of the mark it draw, which indicates the high and low values; the "line" is the horizontal portion of the mark, which indicates the close value.) The HighLowPlot widget also has four standard sets of XiPlotData resources that you can use to specify the high, low, and close data values, and, optionally, an X axis value for each (high, low, close) triple.

These resources are summarized in Table 12-1 and are documented in detail in the subsections below. Figure 12-1 depicts three data points in a HighLowPlot, and shows the resources that control the visual appearance of the "bar" and "line".

12.2.1 Visual Resources

The following resource control the size and color of the lines drawn by the XiHighLowPlot widget.

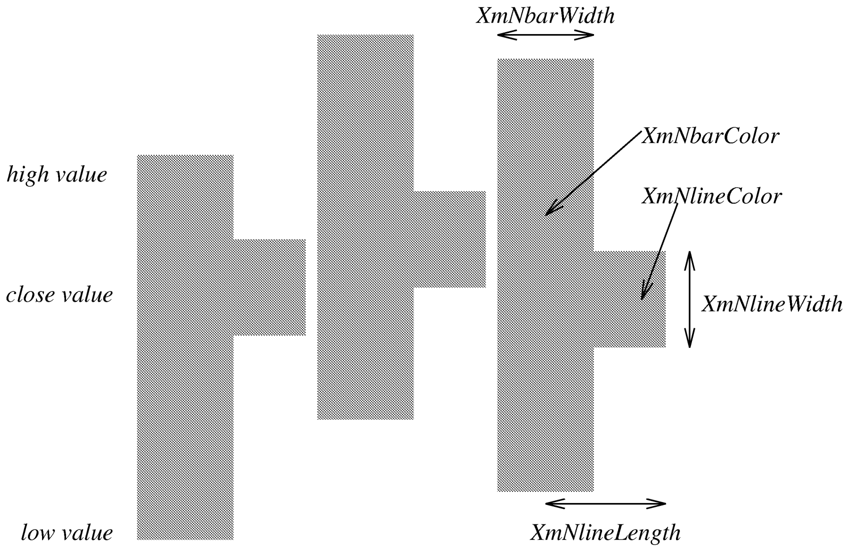


Figure 12-1. Visual resources of the HighLowPlot widget

XmNbarColor

The color of the vertical line drawn between the high and low data values. If this resource is not set, its default value is inherited from the XmNforeground resource of the parent Plotter widget.

XmNbarWidth

The width, in pixels, of the vertical line drawn between the high and low data values. The default value is 5 pixels, which makes this line into a bold vertical "bar".

XmNlineColor

The color of the horizontal "line" drawn to the right of the vertical "bar" to mark the close position in the data. If this resource is not set, it inherits its value from the **XmNforeground** resource of the parent Plotter widget.

XmNlineWidth

The width, in pixels, of the horizontal line drawn to mark the position of the close value. The default is 3 pixels, which is a line wide enough to match the 5 pixel wide bar, but narrow enough to mark the position of the close value with some accuracy.

XmNlineLength

The length, in pixels, of the horizontal line drawn to the right of the bar to mark the position of the close value. The default is 6 pixels. Note, however, that with the default bar width of 5 pixels, the "bar" extends for two pixels on either side of the X point, and only 4 pixels of the 6 pixel-long line will be visible to the right of the bar.

12.2.2 Data Resources

By far the majority of HighLowPlot resources are data resources. Unlike the LinePlot widget which takes an (x, y) pair of values for each data point, the HighLowPlot widget takes an (x, high, low, close) tuple for each data point. The HighLowPlot widget, therefore, has four complete sets of XiLinePlot resources. The resource names, types, and default values are listed in Table 12-1. Chapter 4, *Plot Data Representation* explains how to use the XiLinePlot abstraction, and if you have read that chapter, the meaning of each of these resources will be evident.

Note that there is only a single XmNnumValues resource, not separate resources for the number of high values, the number of low values, and so on. Since there must always be the same number of values specified by each of the XiPlotData structures, a single resource is sufficient. Be careful not to set or change XmNhighValues, XmNlowValues, or any of the other "Values" resources before you set the XmNnumValues resource. If you set XmNhighValues to a new array with fewer values than the old array, and if you did not change XmNnumValues in the same or a previous call, then the widget might attempt to extract values beyond the end of the array you specified, which could result in garbage data, or in a segmentation fault.

Very often when you use the HighLowPlot widget, your data will be uniformly spaced along the X axis. In this case, you need not specify XmNxValues, XmNxValueSize, XmNxValueOffset, and so on. Instead, you can simply specify the first X value, and the interval between the values, with XmNxValueConstant and XmNxValueMultiplier. There are times, however, when there will be gaps in your data (on market holidays, for example, there are no high, low, and close values for stocks, because no trading takes place) and you will want to explicitly specify an array of X values along with your arrays of high, low, and close values.

The HighLowPlot widget can also be used to display (high, low) data without any close values. A typical example would be a plot of daily temperature extremes over a period of a month. In the current implementation of the HighLowPlot widget, you must always specify a value for the XmNcloseValues and related resources, however. Even though you must specify close data, you can prevent that data from being displayed by setting the XmNlineLength resource to 0. And, since the close data will not be visible, you can use any valid array you choose. Perhaps the easiest thing to do is to set the close data resources to the same values as the high data resources--then the XiHighLowPlot widget will extract its valid (but invisible and meaningless) close values from the same place it extracts its (actually meaningful) high values. This technique means that you do not have to create a dummy array of close values.

12.3 Specifying XiHighLowPlot Data

The most general way to specify data for a HighLowPlot widget is through the data resources described in the previous section. It is often easier, however, to use one of the convenience functions provided by the HighLowPlot for this purpose. There are six convenience functions, documented in the subsections below.

12.3.1 Arrays of Doubles, Floats, or Ints

If your data values occur at uniform intervals along the X axis, and you store your high, low, and close values in three separate arrays all of type double, float, or int, then you can set your HighLowPlot data with one of the three functions shown in Figure 12-2.

```
void XiHighLowPlotSetDoubleValues(Widget w,
                                   double xmin, double xinterval,
                                   double *highvals, double *lowvals,
                                   double *closevals,
                                   Cardinal num_points);

void XiHighLowPlotSetFloatValues(Widget w,
                                  double xmin, double xinterval,
                                  float *highvals, float *lowvals,
                                  float *closevals,
                                  Cardinal num_points);

void XiHighLowPlotSetIntValues(Widget w,
                                double xmin, double xinterval,
                                int *highvals, int *lowvals,
                                int *closevals,
                                Cardinal num_points);
```

`w` The HighLowPlot widget that is having its data set.

`xmin` The X value for the first data point.

`xinterval`

The interval between X data values. If your X data points are not evenly spaced, then you cannot use any of these convenience functions.

`highvals, lowvals, closevals`

Arrays of high, low, and close data points. The arrays must each be of the same type (double, float, or int, depending on the function), and must each have the same number of elements.

`num_points`

The number of values in each of the arrays.

Figure 12-2. Specifying data as arrays of doubles, floats, or ints

12.3.2 Using Arrays of Structures

Often it is more convenient for an application to store its high, low, and close values in arrays of complex structures, rather than making a copy of the data into a special array of double, for example. The functions XiHighLowPlotSetValues() and XiHighLowPlotSetPoints(), shown in Figure 12-3 and Figure 12-4 provide the flexibility to do this.

```
void XiHighLowPlotSetValues(Widget w, double xmin, double xinterval,
                            XiPlotDataType ytype, Cardinal ysize,
                            XtPointer highvals, Cardinal highoffset,
                            XtPointer lowvals, Cardinal lowoffset,
                            XtPointer closevals, Cardinal closeoffset,
                            Cardinal num_points);
```

`w` The XiHighLowPlot widget that is having its data set.

`xmin, xinterval`

The X axis coordinate of the first (high, low, close) point, and the X interval between adjacent points.

`ytype, ysize`

These arguments specify the type of the high, low, and close values (the "y values"), and the size of the structures each is stored within.

`highvals, lowvals, closevals`

These arguments are pointers to arrays of structures that contain the high, low, and close data. These may be three different arrays, but often they will all be pointers to the same array. Each element of these arrays must be of the size specified by the ysize argument.

`highoffset, lowoffset, closeoffset`

These arguments are the byte offsets of the high, low, and close value fields within their respective data structures. You can compute these offsets with the standard Xt macro XtOffsetOf().

`num_points`

The number of data points to appear in the graph. Each of the highvals, lowvals, and closevals arrays must have at least this many elements.

Figure 12-3. XiHighLowPlotSetValues()

The difference between XiHighLowPlotSetValues() and XiHighLowPlotSetPoints() is that XiHighLowPlotSetValues() assumes evenly spaced X points, and XiHighLowPlotSetPoints() requires you to specify a complete set of X values. When the X coordinates of your data points are all uniformly spaced, and there are no gaps in your data, you can use the xmin and xinterval arguments of XiHighLowPlotSetValues() to specify X positions without supplying an array of values.

```
void XiHighLowPlotSetPoints(Widget w,
                             XtPointer xvals, XiPlotDataType xtype,
                             Cardinal xsize, Cardinal xoffset,
                             XiPlotDataType ytype, Cardinal ysize,
                             XtPointer highvals, Cardinal highoffset,
                             XtPointer lowvals, Cardinal lowoffset,
                             XtPointer closevals, Cardinal closeoffset,
                             Cardinal num_points);
```

`w` The XiHighLowPlot widget that is having its data set.

`xvals, xtype, xsize, xoffset`

These four arguments are used by XiHighLowPlotSetPoints() to explicitly specify an array of X coordinates for your (high, low, close) data.

`ytype, ysize`

These arguments specify the type of the high, low, and close values (the "y values"), and the size of the structures each is stored within.

`highvals, lowvals, closevals`

These arguments are pointers to arrays of structures that contain the high, low, and close data. These may be three different arrays, but often they will all be pointers to the same array. Each element of these arrays must be of the size specified by the ysize argument.

`highoffset, lowoffset, closeoffset`

These arguments are the byte offsets of the high, low, and close value fields within their respective data structures. You can compute these offsets with the standard Xt macro XtOffsetOf().

`num_points`

The number of data points to appear in the graph. Each of the highvals, lowvals, and closevals arrays must have at least this many elements.

Figure 12-4. XiHighLowPlotSetPoints()

When your data does not have a regular sampling interval, or when there are gaps in the data at which you do not want a bar drawn, you can use XiHighLowPlotSetPoints(), with its xvals, xtype, xsize, and xoffset arguments. These arguments correspond to the XmNxValues, XmNxValueType, XmNxValueSize and XmNxValueOffset resources, and specify a pointer to an array of structures, the type of the value to be extracted from each structure in the array, the size of each structure, and the byte offset of the value within each structure.

Note that both of these functions make the assumption that your high, low, and close values will always be stored in variables of the same type in structures of the same size. This is a fairly reasonable assumption, since you will usually use this function when you have stored your values together in the same structure. This type and size are specified by the ytype and ysize arguments.

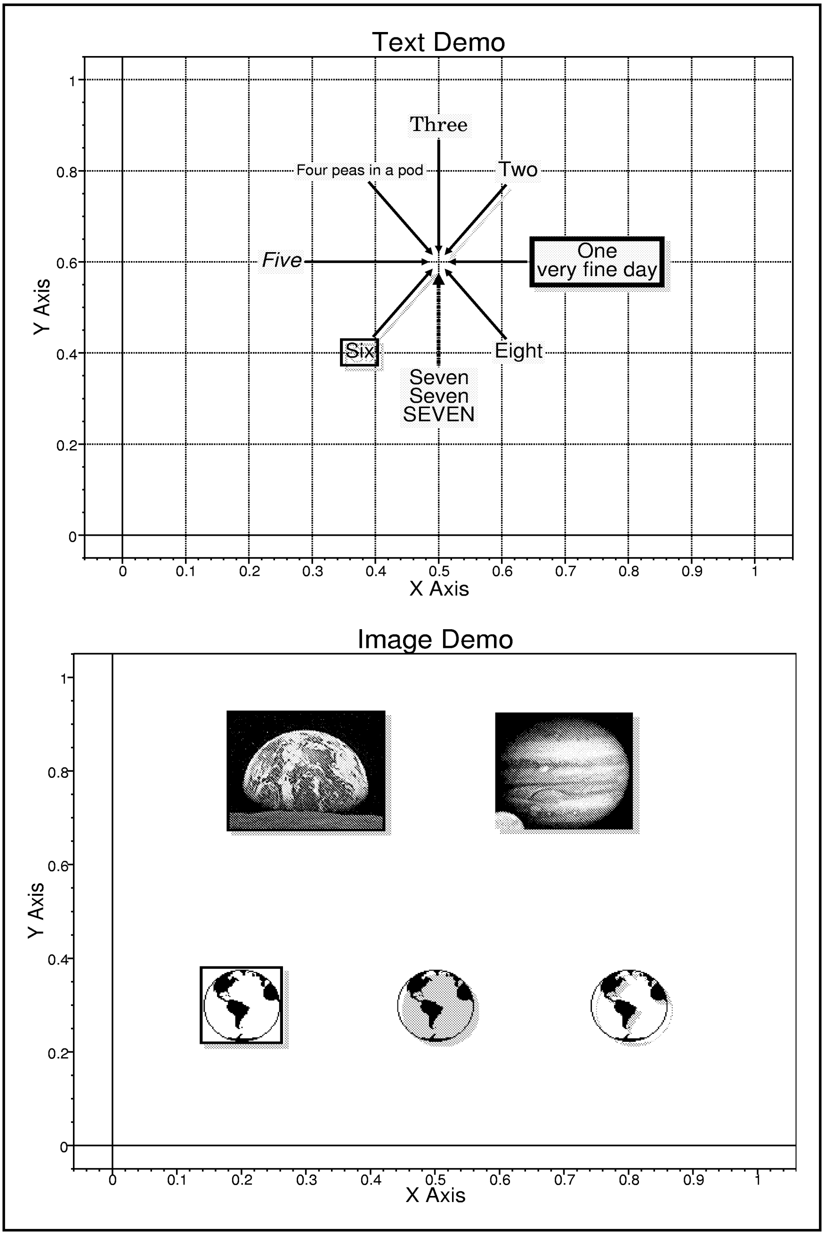
12.3.3 Notifying the Widget of Changed Data

Since the HighLowPlot widget extracts its data values directly from application data structures, it needs to be notified when that data has changed underneath it. Some applications will define static data structures, and leave them unchanged for the lifetime of the program. Other applications, however will update plot data dynamically. One way to do this is to give the widget a pointer to new data arrays, and then free the memory associated with the old ones. Another approach is to write the new plot data into the existing arrays, and notify the widget that its data has changed and that it needs to update itself. You can do this with XiHighLowPlotDataChanged(), shown in Figure 12-5.

```
void XiHighLowPlotDataChanged(Widget w);
```

`w` The HighLowPlot widget that has had its data changed. Calling this function will cause that widget to extract and display new data from the application data structures.

Figure 12-5. XiHighLowPlotDataChanged()


Text Demo
Three
Four peas in a pod
Two
Five
One
very fine day
Six
Eight
Seven
Seven
SEVEN
Y Axis
X Axis
1
0.8
0.6
0.4
0.2
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Image Demo
Y Axis
X Axis
1
0.8
0.6
0.4
0.2
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

13

Annotating Plots

The TextPlot and ImagePlot widgets behave somewhat like a Motif XmLabel widget--they allow you to display arbitrary text (the TextPlot) or graphics (the ImagePlot) within the Plotter widget. These textual or graphical "annotations" are associated with a particular position in the plotter (an axis position, pixel position, or percentage position) and either appear at that position, or are offset from it slightly, with an arrow drawn from the annotation to the point that is being annotated.

TextPlot Synopsis

Class Name: XiTextPlot

Class Hierarchy: RectObj → XiPlot → XiTextPlot

Header File: *<Xi/TextPlot.h>*

Class Pointer: `xiTextPlotWidgetClass`

Constructor:

```
XiCreateTextPlot (Widget parent, String name,
                  ArgList args,
                  Cardinal num_args);
```

ImagePlot Synopsis

Class Name: XiImagePlot

Class Hierarchy: RectObj → XiPlot → XiImagePlot

Header File: *<Xi/ImagePlot.h>*

Class Pointer: `xiImagePlotWidgetClass`

Constructor:

```
XiCreateImagePlot (Widget parent,String name,
                   ArgList args,
                   Cardinal num_args);
```

Both the TextPlot and the ImagePlot widget are subclasses of the AnnotationPlot widget class. While you will never use the AnnotationPlot directly, it provides the framework for positioning and offsetting the annotation, drawing an arrow to the annotated point, and drawing an optional border around the annotation, and an optional shadow beneath it.

Text annotations, implemented with the TextPlot widget, can be used to "mark up" a display, providing additional information about a point, a plot, or the entire Plotter. For example, you can use a TextPlot widget to describe the significance of a particular data point, or as a caption or sub-title for a Plotter. The TextPlot widget supports multi-line text using the same wide range of fonts, styles, and point sizes supported by the Plotter and axes.

The ImagePlot widget allows graphical annotations in the Plotter--you can use ImagePlots to add logos and icons to your display. A clip mask can be provided to display non-rectangular images.

The following sections demonstrate how to create TextPlot and ImagePlot annotations, explain each of the resources of these widgets, and document functions for easily positioning and updating annotations.

13.1 Creating Annotations

You can create text and graphical annotations with the functions XiCreateTextPlot(), and XiCreateImagePlot() which are standard Motif-style widget constructors. If you prefer to use XtCreateWidget() or XtVaCreateWidget(), the class pointer for the TextPlot widget is xiTextPlotWidgetClass, and the class pointer for the ImagePlot widget is xiImagePlotWidgetClass. These constructor functions and widget class pointers are declared in the header files <*Xi/TextPlot.h*> and <*Xi/ImagePlot.h*>.

Example 13-1 demonstrates how you might create a text annotation and an image annotation. Don't forget that you must manage any widgets you create with XtManageChild() or XtManageChildren().

Example 13-1. Creating annotations

```
#include <Xi/Plotter.h>   /* always include this */
#include <Xi/TextPlot.h> /* declares constructor and class pointer
*/
#include <Xi/ImagePlot.h> /* ditto */
#include "my_bitmap.xbm"  /* bitmap data created with the 'bitmap'
client */

Widget plotter; /* assume this is already created */
Widget textplot, imageplot;
Bitmap my_bitmap;
Arg args[10];
int i;

/* Create a text plot */
i = 0;
XtSetArg(args[i], XmNtext, "My first annotation"); i++;
XtSetArg(args[i], XmNxPercent, 50); i++;
XtSetArg(args[i], XmNyPercent, 50); i++;
XtSetArg(args[i], XmNxPercentOffset, -25); i++;
XtSetArg(args[i], XmNyPercentOffset, -25); i++;
textplot = XiCreateTextPlot(plotter, "textplot", args, i);

/* Create the bitmap for display by the image plot */
my_bitmap = XCreateBitmapFromData(XtDisplay(plotter), XtWin-
dow(plotter),
                                  my_bitmap_bits,
                                 my_bitmap_width, my_bitmap_height);

/* Create the image plot */
imageplot = XtVaCreateWidget("imageplot", xiImagePlotWidgetClass,
plotter,
                             XmNbitmap, my_bitmap,
                             XmNdrawShadow, True,
                             XmNxPercent, 50,
                             XmNyPercent, 50,
                             XmNxPercentOffset, 25,
                             XmNyPercentOffset, 25,
                            XtVaTypedArg, XmNforeground, XmRString,
"red", 4,
                             XtVaTypedArg, XmNbackground, XmR-
String,"green",6,
                             NULL);

/* the plots must be managed before they are visible */
XtManageChild(textplot);
XtManageChild(imageplot);
```

13.2 TextPlot and ImagePlot Resources

The TextPlot and ImagePlot widgets are both subclasses of the AnnotationPlot widget, and most of their resources come from this superclass. The AnnotationPlot has lots of resources that control the position and size of the annotation, resources that specify whether the annotation should have a border, an arrow, or a shadow, and resources that specify the colors and other attributes of the border, arrow, and shadow.

The TextPlot widget only has a few resources of its own; these specify the text to be displayed, its justification, and the font it is to be displayed in. Similarly, the ImagePlot defines only a few new resources to specify the bitmap, pixmap or XImage to be displayed, and an optional clip-mask for non-rectangular images.

Table 13-1 summarizes the resources of these three related widget classes, and the subsections that follow explain each resource in more detail. For convenience, the table of resources also appears in the quick reference section at the end of this manual.

Table 13-1. AnnotationPlot Resources

Name	Class	Type	Default	CSG	Pg
Position					
XmNxPercent	XmCXPercent	XmRInt	*Computed*	CSG	184
XmNyPercent	XmCYPercent	XmRInt	*Computed*	CSG	184
XmNxPixel	XmCXPixel	XmRPosition	0	CSG	185
XmNyPixel	XmCYPixel	XmRPosition	0	CSG	185
XmNxPoint	XmCXPoint	XmRDouble	*Computed*	CSG	185
XmNyPoint	XmCYPoint	XmRDouble	*Computed*	CSG	185
Offset					
XmNxPercentOffset	XmCPercentOffset	XmRInt	*Computed*	CSG	186
XmNyPercentOffset	XmCPercentOffset	XmRInt	*Computed*	CSG	186
XmNxPixelOffset	XmCPixelOffset	XmRPosition	0	CSG	185
XmNyPixelOffset	XmCPixelOffset	XmRPosition	0	CSG	185
XmNadjustOffset	XmCAdjustOffset	XmRBoolean	TRUE	CSG	186
Size and margins					
XmNmarginWidth	XmCMargin	XmRDimension	2	CSG	187
XmNmarginHeight	XmCMargin	XmRDimension	2	CSG	187
XmNboxWidth	XmCWidth	XmRDimension	*Computed*	CSG	187
XmNboxHeight	XmCHeight	XmRDimension	*Computed*	CSG	187
Colors					
XmNforeground	XmCForeground	XmRPixel	XmNforeground[a]	CSG	187
XmNbackground	XmCBackground	XmRPixel	XmNbackground[b]	CSG	187
Arrows					
XmNdrawArrow	XmCDrawArrow	XmRBoolean	TRUE	CSG	188
XmNarrowColor	XmCArrowColor	XmRPixel	XmNforeground[a]	CSG	189
XmNarrowShaftPattern	XmCArrowShaftPattern	XmRLinePattern	NULL	CSG	189
XmNarrowShaftWidth	XmCArrowShaftWidth	XmRDimension	2	CSG	189
XmNarrowHeadWidth	XmCArrowHeadSize	XmRDimension	6	CSG	189

XmNarrowHeadLength	XmCArrowHeadSize	XmRDimension	6	CSG	189
XmNarrowHeadDistance	XmCArrowDistance	XmRDimension	4	CSG	189
XmNarrowTailDistance	XmCArrowDistance	XmRDimension	2	CSG	189
Shadows and Trim					
XmNdrawShadow	XmCDrawShadow	XmRBoolean	FALSE	CSG	189
XmNdrawArrowShadow	XmCDrawShadow	XmRBoolean	FALSE	CSG	190
XmNshadowColor	XmCForeground	XmRPixel	XmNforeground[a]	CSG	190
XmNstippleShadow	XmCStippleShadow	XmRBoolean	TRUE	CSG	190
XmNshadowXOffset	XmCShadowOffset	XmRPosition	4	CSG	190
XmNshadowYOffset	XmCShadowOffset	XmRPosition	4	CSG	190
XmNtrimColor	XmCForeground	XmRPixel	XmNforeground[a]	CSG	190
XmNtrimWidth	XmCTrimWidth	XmRDimension	0	CSG	190
TextPlot Resources					
XmNfontList	XmCFontList	XmFontList	XmNfontList[d]	CSG	191
XmNpsFontList	XmCPSFontList	XiPSFontList	XmNpsFontList[e]	CSG	191
XmNtextString	XmCTextString	XmString	NULL[c]	CSG	191
XmNtext	XmCText	XmRString	NULL	CSG	191
XmNfontSize	XmCFontSize	XmRDimension	12	CSG	191
XmNfontStyle	XmCFontStyle	XmRString	NULL	CSG	191
XmNalignment	XmCAlignment	XmRAlignment	XiALIGNMENT_CENTER	CSG	192
XmNstringDirection	XmCStringDirection	StringDirection	*dynamic*	CSG	192
XmNopaque	XmCOpaque	XmRBoolean	TRUE	CSG	192
ImagePlot Resources					
XmNpixmap	XmCPixmap	XmRPixmap	None	CSG	192
XmNbitmap	XmCBitmap	XmRPixmap	None	CSG	192
XmNimage	XmCImage	XmRXImage	NULL	CSG	192
XmNclipMask	XmCClipMask	XmRPixmap	None	CSG	193

[a] *Inherited from the Plotter XmNforeground resource.*

[b] *Inherited from the Plotter XmNbackground resource.*

[c] *When an XmString resource is unset, its value is created from the corresponding String resource, using a font tag derived from the corresponding style and size resources.*

[d] *Inherited from the Plotter XmNfontList resource.*

[e] *Inherited from the Plotter XmNpsFontList resource.*

13.2.1 Annotation Position

A traditional plot depicts a collection of (x,y) values. By contrast, an annotation plot is associated with only one (x,y) pair, representing the point being annotated. By default an annotation will be centered over the point it annotates. (Alternatively, the annotation can be offset an arbitrary distance from the annotated point and an arrow can be drawn from the annotation to the point annotated--offsets and arrows will be discussed later in this chapter.)

The AnnotationPlot widget provides three different ways to specify the point that is to be annotated: in axis coordinates, pixel coordinates, or as a percentage of the size of the plotting area. The following paragraphs explain the resources you use to set an annotation position in each of these ways, and Figure 13-1 shows text annotations positioned in each of these ways. Note that none of the resources documented below are name XmNxPosition or XmNyPosition--those are PiePlot resources, not AnnotationPlot resources!

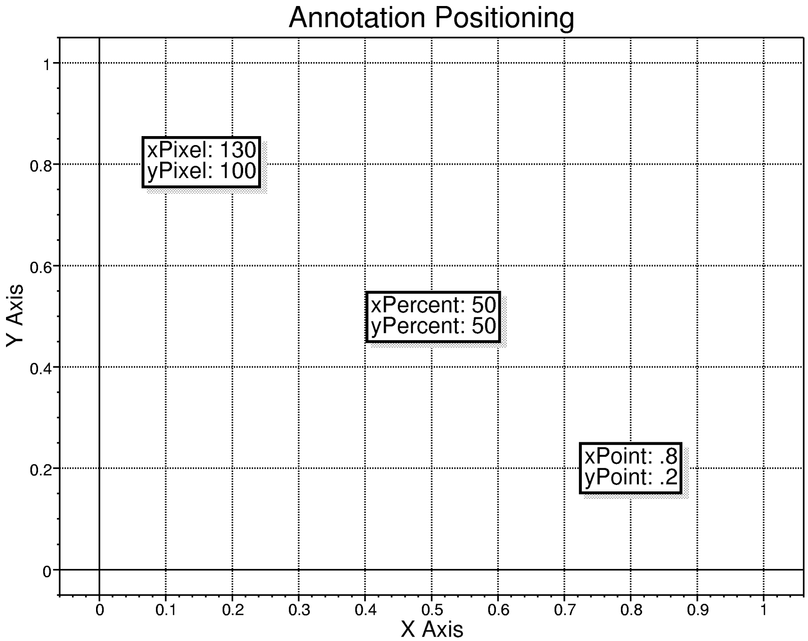


Figure 13-1. Positioning annotations

XmNxPercent, XmNyPercent

The horizontal and vertical positions of the point being annotated, as a percentage of the width and height of the Plotter. These resources are integers and should be set to a value between 0 and 100. Annotations positioned with these resources will remain at a fixed pixel position when the axis bounds change, and will remain at the same proportional position in the plotting area when the Plotter widget is resized. When these resources are set, the values of the XmNxPixel, XmNyPixel, XmNxPoint, and XmNyPoint resources will be computed to match (i.e. those resources can be meaningfully queried.)

XmNxPixel, XmNyPixel

The X and Y coordinates of the point being annotated, measured in pixels from the upper left corner of the plot area. Annotations positioned with these resources will remain at the same absolute position in the plotting area regardless of the axis bounds and the size of the Plotter. Annotations positioned this way may be suitable for titles or other displayed text. When these resources are set, the values of the other positing resources will be computed to match.

XmNxPoint, XmNyPoint

The X and Y position of the point to be annotated, in axis coordinates. Annotations positioned with these resources will move when the axis bounds or the Plotter size change--this makes them ideal for annotating a particular data point. When these resources are set, the values of the other position resource will be computed to match.

13.2.2 Annotation Offsets

If you would like an arrow drawn from the annotation body to the point being annotated, you must specify horizontal and vertical offsets for the annotation. These offsets represent the distance from the annotated point to the center of the annotation; the arrow will be drawn in the intervening space. Offsets may be negative. If you do not want an arrow drawn between the annotation and the annotated point, there is no real reason to specify an offset. Instead, add the desired offset to the annotation's position resources and leave its offsets at their default of zero.

You can specify an offsets either in pixels or as a percentage of the width and height of the plotting area. The following paragraphs explain the applicable resources, and Figure 13-2 shows three TextPlot widgets that annotate the same point but have different offsets. In addition, one additional resource, XmNadjustOffset specifies whether offsets should be adjusted, when necessary, to ensure that the annotation fits entirely within the plotting area.

XmNxPixelOffset, XmNyPixelOffset

The horizontal and vertical distance, in pixels, from the annotated point to the center of the annotation. If these resources are set, the value of the XmNxPercentOffset and XmNyPercentOffset resources will be computed to match.

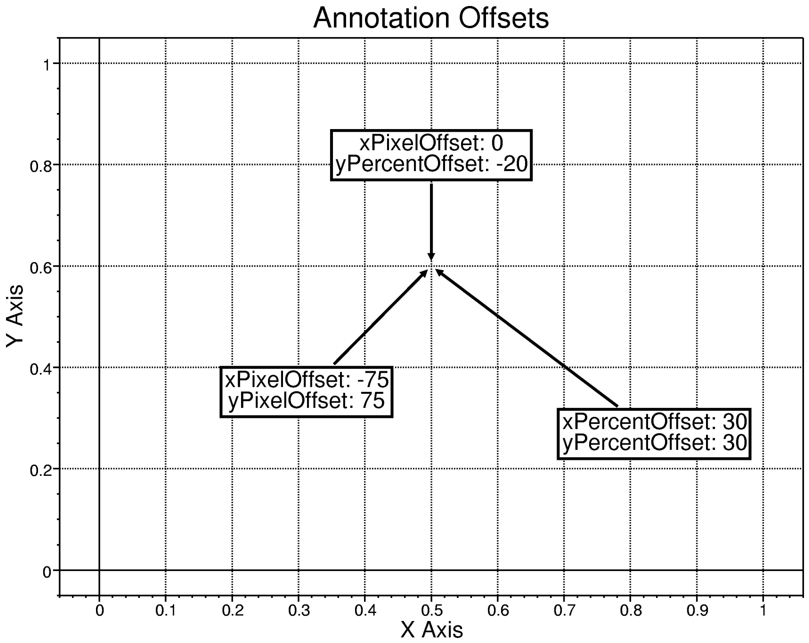


Figure 13-2. Annotation offsets

XmNxPercentOffset, XmNyPercentOffset

The horizontal and vertical distance from the annotated point to the center of the annotation, measured as a percentage of the width of the Plotter. These resources are integers and should be set to a value between 0 and 100. If these resources are set, the values of the XmNxPixelOffset and XmNyPixelOffset resources will be adjusted correspondingly.

XmNadjustOffset

Whether the annotation's offset should be adjusted if necessary to insure that it fits completely within the plot area. If this resource is True (the default), then you can be sure that your annotation will not move off the edge of the screen even if the axis bounds or the plotter size change. With this resource you can also position an annotation flush against an edge of the plot area by specifying a location that would otherwise cause all or part of the annotation to appear beyond that edge.

13.2.3 Annotation Size and Margins

The overall size of an annotation is, by default, its base size (i.e. the size required to display its text or its image) plus its margins, plus the width of its "trim" (i.e. it border.) The resources described in this section allow you to set the margin size for an annotation, and also provide a way to set a fixed absolute size for an annotation, regardless of its base size.

XmNmarginWidth, XmNmarginHeight

XmNmarginWidth specifies the distance, in pixels, from the annotation's border to its left and right edges. XmNmarginHeight specifies the distance, in pixels, from the annotation's border to its top and bottom edges.

XmNboxWidth, XmNboxHeight

You can query these resources to determine the overall width and height, in pixels, of the annotation. Normally, you will not set these resources-- instead, you can adjust the annotation's dimensions with the XmNmarginWidth and XmNmarginHeight resources, which control the amount of space outside of the annotation's contents. If you do set these resources directly, it will specify a fixed size for the annotation that is not dependent on the base size of the annotation's text or image. In this case, the annotation's margins will be adjusted as necessary to provide the specified size. If you set the XmNboxWidth or XmNboxHeight resource to zero (the default), the annotation will be drawn at its default size (based on the size of the text or image) and these resources will be changed to reflect that default size.

13.2.4 Annotation Colors

The XmNforeground and XmNbackground resources described in this section specify only the color of the annotation itself--i.e. the text or image of the annotation. The colors of the annotation's arrow, trim, and shadow are specified by other resources, described elsewhere.

XmNforeground, XmNbackground

These are AnnotationPlot resources, but are not used directly by the Annotation plot--instead they are provided for use by the AnnotationPlot subclasses. For the TextPlot, they specify the foreground color of the text to display, and the background color (for opaque TextPlots) of the box that surrounds the text. For the ImagePlot, they specify the foreground and background color of bitmaps. (Images specified as Pixmaps or XImages do not use these resources.) If you do not set these resources, their values are inherited from the XmNforeground and XmNbackground resources of the parent Plotter widget.

13.2.5 Arrows

Annotations can be drawn with an optional arrow between the body of the annotation and the annotated point. By default, an arrow will be drawn whenever there is sufficient space--i.e. whenever the X and Y offsets are large enough. To disable the arrow altogether, set the XmNdrawArrow resource to False. The resources described in this section control the appearance of the arrow.Figure 13-3 shows some arrow variations.

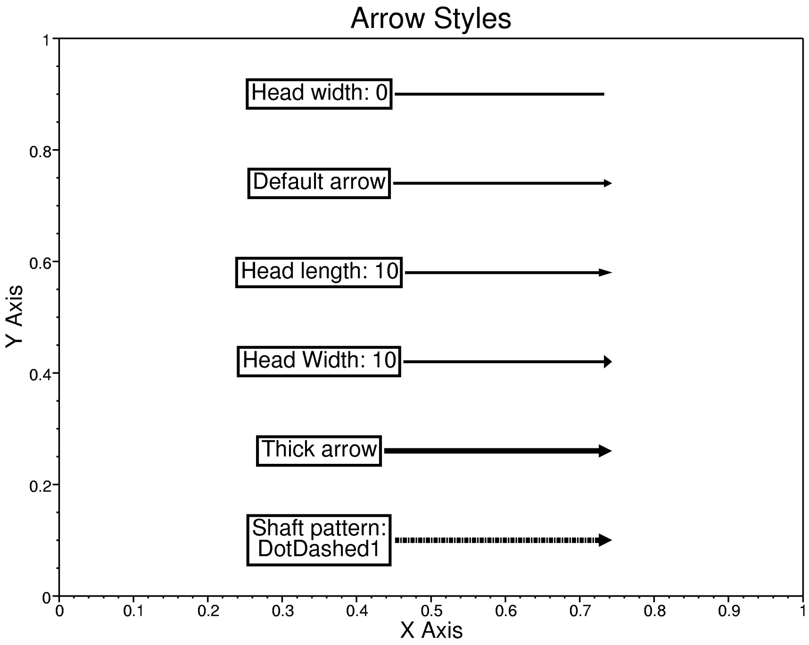


Figure 13-3. Annotation arrow styles

XmNdrawArrow

A Boolean that specifies whether to draw an arrow, if possible, between the annotation and the point being annotated. If there is not enough space between the annotation and the annotated point, no arrow will be drawn. The default is True.

XmNarrowColor

The color to use for the annotation's arrow. If you do not set this resource, its value is inherited from the XmNforeground resource of the parent Plotter widget.

XmNarrowShaftPattern

The line pattern to use for the arrow shaft. The default is NULL, which specifies a solid shaft. Chapter 3, *Plotter Data Types and Resources* explains how to specify line patterns.

XmNarrowShaftWidth

The width in pixels of the arrow shaft.

XmNarrowHeadWidth

The width in pixels of the arrow head. If set to zero, no arrow head will be drawn. In general, the value of this resource should be less than or equal to that of the XmNarrowHeadLength resource, and should be no more than three times larger than the XmNarrowShaftWidth.

XmNarrowHeadLength

The distance in pixels from the tip of the arrow head to the shaft. If set to zero, no arrow head will be drawn. In general, the value of this resource should be greater than or equal to that of the XmNarrowHeadWidth resource.

XmNarrowHeadDistance

The distance in pixels from the tip of the arrow to the point being annotated. You may wish to increase the value of this resource to prevent the arrow head from overlapping a marker or a particularly thick line.

XmNarrowTailDistance

The distance in pixels from the tail of the arrow to the nearest edge of the annotation, measured in the direction of the arrow shaft.

13.2.6 Shadows and Trim

The resources in this section control the appearance of the shadow optionally drawn beneath the annotation and the appearance of the "trim" or border optionally drawn around the annotation.

XmNdrawShadow

A Boolean that specifies whether the AnnotationPlot widget will draw shadows beneath its contents and its trim. The default is False.

XmNdrawArrowShadow

A Boolean that specifies whether to draw a shadow under the annotation's arrow. The default is False.

XmNshadowColor

The color of the shadow to be drawn. If you do not set this resource, its value is inherited from the XmNforeground resource of the parent Plotter widget. Note that if XmNstippleShadow is True, then the stippling will result in an apparent shadow color that is significantly lighter than the value specified by this resource.

XmNstippleShadow

A Boolean that specifies whether the annotation's shadows should be drawn with a solid color or with a stippled color. The default is True which specifies that a stipple should be used. Note that on monochrome systems, stippling is required because the shadow color and the line color will be the same.

XmNshadowXOffset, XmNshadowYOffset

The offset, in pixels, of the shadow from the annotation in the X and Y dimensions. The default for each of these resources is 2 pixels. Larger values will produce a shadow further away from the actual annotation, increasing the apparent three dimensional height of the annotation above the Plotter background. Note that negative values are legal for this resource, and simply change the position of the imaginary light source that is casting the shadow.

XmNtrimColor

The color of the annotation's trim (i.e. border). If you do not set this resource, its value is inherited from the XmNforeground resource of the parent Plotter widget.

XmNtrimWidth

The width in pixels of the annotation's trim (i.e. the rectangle drawn around the border of the annotation). If this resource is set to 0, then no trim will be drawn.

13.2.7 TextPlot Resources

The resources described in this section are TextPlot resources, and are not applicable to the ImagePlot annotation type. They specify the text to appear in the annotation, and how that text should be drawn. Note that the AnnotationPlot XmNforeground and XmNbackground resources specify the color of the text annotation.

XmNfontList

The font or fonts in which the TextPlot textual annotation should be displayed. If this resource is unspecified, its value is inherited from the parent Plotter's XmNfontList resource.

XmNpsFontList

The list of PostScript fonts to be used to display the TextPlot's textual annotation in hardcopy. If no value is specified for this resource, its default value is inherited from the parent Plotter XmNpsFontList resource. You need only specify this resource if XmNfontList contains fonts outside of the "standard" font families--Adobe Times, Helvetica, Courier, New Century Schoolbook, and Symbol. See Section 3.3, *Fonts and Displayed Text* for information on specifying PostScript font lists.

XmNtextString

The XmString to be displayed by the TextPlot widget. If no XmString is specified for this resource, then one will be created using the XmNtext string, and a font list tag constructed from the XmNfontStyle and XmNfontSize resources, as described in Section 3.3, *Fonts and Displayed Text.* Setting this resource will override any value previously created from the XmNtext string. If you query this resource, the Plotter returns its private, internal copy of the XmString, which you must not modify or free.

XmNtext

The text to appear in the annotation, if XmNtextString is not specified. Note that this resource is an ordinary null-terminated character string, not a Motif XmString. When you set this resource, the string is converted to an XmString using a font list tag constructed from the XmNfontStyle and XmNfontSize resources, and this newly created XmString becomes the new value of the XmNtextString resource. See Section 3.3 for details. When you set this resource, the TextPlot widget makes a private copy of the string, so the application may modify or free its copy of the string without affecting the widget. When you query this resource, you obtain this internal copy, which you must not modify or free.

XmNfontStyle

The typeface of the annotation. This value is used in the font list tag of the XmNtext resource, if no XmNtextString is specified. The default is NULL. When you set this resource, the string value is copied, and when you query it, you must not modify or free the returned value.

XmNfontSize

The point size of the annotation. This value is used in the font list tag of the XmNtext, if no XmNtextString is specified. The default is 12.

XmNalignment

The justification of text that spans multiple lines. Possible values are XmALIGNMENT_BEGINNING, XmALIGNMENT_CENTER, and XmALIGNMENT_END. The default is XmALIGNMENT_CENTER.

XmNstringDirection

This resource is just like the XmLabel widget resource of the same name--it tells the TextPlot whether the XmString it is to display is a left-to-right string or a right-to-left string. Possible values for this resource are XmSTRING_DIRECTION_L_TO_R and XmSTRING_DIRECTION_R_TO_L. The default value depends on the locale. Note that setting this resource does not affect the direction that the text is displayed in; it merely controls the interpretation of the XmNalignment resource value.

XmNopaque

If True (the default), the text will be drawn in a rectangular region filled with the XmNbackground color. If False, the text will be drawn directly onto the plot area, without erasing anything beneath it.

13.2.8 ImagePlot Resources

The resources described in this section are ImagePlot resources, and are not applicable to the TextPlot widget. Three of these resources allow three different ways of specifying the image to be displayed--as a bitmap, a pixmap, or as an XImage. The fourth resource specifies a clip mask to be used to display non-rectangular image annotations.

XmNpixmap

A Pixmap to draw. The depth of the Pixmap must match depth of Plotter window. Call XiImagePlotUpdate() to update the ImagePlot if you change the contents of the Pixmap without changing the value of the resource itself.

XmNbitmap

A bitmap to draw. Set bits in the bitmap will be drawn in the AnnotationPlot XmNforeground color, and unset bits will be drawn in the XmNbackground color. Call XiImagePlotUpdate() to update the ImagePlot if you change the contents of the bitmap without changing the value of the resource itself.

XmNimage

An XImage to draw. The depth of the XImage must match depth of Plotter window. Call XiImagePlotUpdate() to update the ImagePlot if you change the contents of the XImage without changing the value of the resource itself.

XmNclipMask

A bitmap to use as a clipping mask when drawing the image. This bitmap must have the same width and height as the image, and is used as a stencil when the image is rendered: the mask and image will be aligned, and only those pixels in the image which fall under bits that have been set in the mask will be drawn. For example, you could use a clip mask containing a filled circle to exclude the pixels in a (rectangular) bitmap, pixmap or XImage that fall outside of the circle. If you set the XmNbitmap and XmNclipMask resources of an ImagePlot widget to the same bitmap, then only the set pixels of the bitmap will be drawn, in the widget's foreground color. Full support for clipping masks is available for on-screen displays. Clipped bitmaps can be printed as they are displayed; however, clipped pixmaps and XImages will be printed in their unclipped (rectangular) form.

13.3 Positioning Annotations

The most awkward detail of working with TextPlot and ImagePlot annotations is positioning them appropriately on the display. This section documents convenience functions that make it easier to position annotations, and in particular, makes it easier to annotate particular points or elements of a LinePlot, BarPlot, PiePlot, or HighLowPlot.

When annotating data points, you'll almost always want to position your annotation using axis coordinates, using the XmNxPoint and XmNyPoint resources. When setting the offset for an annotation, XmNxPercentOffset and XmNyPercentOffset are often more useful than XmNxPixelOffset and XmNyPixelOffset, because then the size of the offset is proportional to the actual size of the Plotter widget. The function XiAnnotationPosition(), shown in Figure 13-4 provides a convenient way to set these four resources. In addition, it addresses another common difficulty in positioning annotations: the decision about which side of the point the annotation should placed on. XiAnnotationPosition() makes the assumption that there is more likely to be important data near the center of the plotting area, and always attempts to position the annotation on the "outside side" of the data point. The exception is when the point to be annotated is close enough to the edge of the plotting area that the annotation would not fit on the "outside". In this case, the annotation is placed towards the inside.

```
void XiAnnotationPosition(Widget annotation, double x, double y,
                           int xoffset, int yoffset)
```

`annotation`

The TextPlot or ImagePlot widget that is to be positioned.

`x, y` The X and Y axis coordinates of the point that is being annotated. These values will be set on the XmNxPoint and XmNyPoint resources of the annotation widget.

`xoffset, yoffset`

The X and Y offset of the annotation from the point being annotated, expressed as a percentage of the Plotter width and height. The absolute value of these arguments is used, and in both the X and Y dimensions, the annotation is offset towards the closest edge of the Plotter (i.e. away from the center of the plotting area) unless it won't fit, in which case it is offset towards the center of the plotting area.

Figure 13-4. XiAnnotationPosition()

The most common use of TextPlot and ImagePlot widgets is to annotate data points in a LinePlot or HighLowPlot, bars in a BarPlot, or wedges in a PiePlot. The functions shown in the following subsections provide a way of obtaining the coordinates of these data points, bars, and lines, so that they can be annotated. These functions are designed to work well with XiAnnotationPosition().

13.3.1 Querying LinePlot and HighLowPlot values

XiLinePlotGetPoint() returns the X and Y coordinates (the axis coordinates, not the pixel coordinates) of a specified point in a LinePlot. Similarly, XiHighLowPlotGetPoint() returns the X coordinate, and the high, low, and close values of a HighLowPlot. The signatures of these functions are shown in Figure 13-5.

```
Boolean XiLinePlotGetPoint(Widget w, int point,
                           double *x_return, double *y_return)

Boolean XiHighLowPlotGetPoint(Widget w, int point,
                              double *x_return, double *high_return,
                             double *low_return, double *close_return)
```

`w` A LinePlot widget or an HighLowPlot widget.

`point` The index of the data point whose coordinates are to be returned. Pass 0 to query the first data point, 1 to query the second, and so on.

`x_return`

For both LinePlots and HighLowPlots, this argument returns the X coordinate of the specified point.

`y_return`

For LinePlots, this argument returns the Y coordinate of the specified point.

`"high_return, low_return, close_return`

For HighLowPlots, these arguments return the Y coordinate of the high, low, and close values for the specified data point.

Figure 13-5. XiLinePlotGetPoint() and XiHighLowPlotGetPoint()

Note that you can query the X and Y coordinates of an ErrorPlot with XiLinePlotGetPoint(). There is no way to query the values of the error bars in an ErrorPlot, however. You'll have to do this by examining your plot data manually.

13.3.2 Querying the Bounds of a BarPlot Bar

XiBarPlotGetBar() is similar in function to XiLinePlotGetPoint(), but is more complicated, because it returns the coordinates of a two-dimensional bar, rather than of a zero-dimensional point. The signature for this function is shown in Figure 13-6.

```
Boolean XiBarPlotGetBar(Widget w, int bar,
                        double *x0_return, double *y0_return,
                        double *x1_return, double *y1_return)
```

`w` A BarPlot widget.

`bar` The index of the bar that is to have its coordinates queried. Pass 0 to query the first bar in the BarPlot, 1 to query the second, and so on.

`x0_return`

This argument returns the left edge of the bar.

`y0_return`

This argument returns the origin of the bar plot, or the bottom edge of a stacked bar. Note that when the value displayed by the bar is positive, the origin is the lower edge of the bar. But when the bar displays a negative value, it extends beneath the origin, and the origin returned by this argument is the upper edge of the bar.

`x1_return`

This argument returns the right edge of the bar. It is always greater than the value returned by *x0_return*.

`y1_return`

This argument returns the value displayed by the bar, or the top edge of a stacked bar. When the bar displays a positive value, this argument will return a larger value than that returned by *y0_return*.

Figure 13-6. XiBarPlotGetBar()

XiBarPlotGetBar() returns the coordinates of two corners of a bar. You can use these values to position your annotation wherever desired. You might position it centered within the bar, for example, or slightly above the top of the bar. Note that you can use XiBarPlotGetBar() to query the bar values for a HistoPlot widget as well.

13.3.3 Querying the Bounds of a PiePlot Wedge

Querying the coordinates of a wedge in a PiePlot is trickier than querying a BarPlot, simply because a wedge is irregularly shaped. XiPiePlotGetPosition() returns the coordinates of several interesting points on a PiePlot widget. (Recall that each wedge of a pie chart is an individual PiePlot widget; thus there is no need to specify which wedge is being queried within a PiePlot.) The signature of this function is shown in Figure 13-7.

```
void XiPiePlotGetPosition(Widget pie,
              double *x0_return, double *y0_return, /*vertex */
              double *x1_return, double *y1_return, /*start of arc */
             double *x2_return, double *y2_return, /*end point of arc */
             double *x3_return, double *y3_return, /* mid-point of arc*/
             double *percent_return)              /* wedge percentage */
```

`pie` A PiePlot widget; a single wedge in a piechart.

`x0_return, y0_return`

These arguments return the X and Y coordinates of the vertex of the wedge.

`x1_return, y1_return`

These argument return the X and Y coordinates of the beginning of the arc of the wedge.

`x2_return, y2_return`

These argument return the X and Y coordinates of the end of the arc of the wedge.

`x3_return, y3_return`

These argument return the X and Y coordinates of the mid-point of the arc.

`percent_return`

This argument returns the percentage of the entire pie that this wedge represents. You might want to convert this value to a string and use it in your annotation.

Figure 13-7. XiPiePlotGetPosition()

You might choose to use the return values of XiPiePlotGetPosition() directly and position your annotation offset slightly from the mid-point of the arc of the wedge, returned by *x3_return* and *y3_return*, for example. Or you might do a little computation on your own and position your annotation with zero offset two-thirds of the way along the line between the vertex and the midpoint of the wedge.

13.3.4 Example: Annotating a PiePlot

Figure 13-2 shows a C procedure that uses XiAnnotationPosition() and XiPiePlotGetPosition() to annotate four different points on a PiePlot wedge. You could write similar code to annotate data in a LinePlot, BarPlot, or HighLowPlot. Figure 13-8 t has had three of its wedges labeled with the code in this procedure. Note how the annotations are always offset towards the "outside" of the plotting area from the point they annotate--this is the work of XiAnnotationPosition().

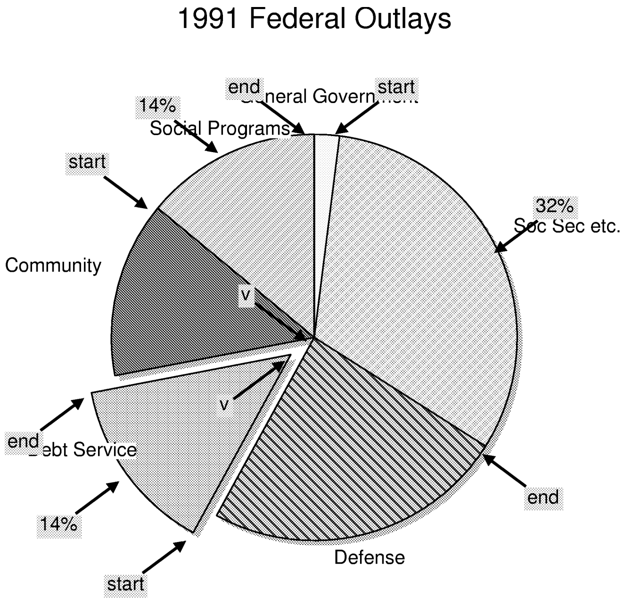


Figure 13-8. A thoroughly annotated pie chart

Example 13-2. Thoroughly annotating a PiePlot wedge

```
void AnnotateWedge(Widget pie)
{
    double x0, x1, x2, x3, y0, y1, y2, y3, percent;
    char percent_string[20];
    Widget annos[4];

    XiPiePlotGetPosition(pie, &x0, &y0, &x1, &y1,
                          &x2, &y2, &x3, &y3, &percent);
    sprintf(percent_string, "%d%%", (int)percent);

    annos[0] = XtVaCreateWidget("anno", xiTextPlotWidgetClass,
                       Plotter, XmNtext, "v", NULL);
    annos[1] = XtVaCreateWidget("anno", xiTextPlotWidgetClass,
                       Plotter, XmNtext, "start", NULL);
    annos[2] = XtVaCreateWidget("anno", xiTextPlotWidgetClass,
                       Plotter, XmNtext, "end", NULL);
    annos[3] = XtVaCreateWidget("anno", xiTextPlotWidgetClass,
                       Plotter, XmNtext, percent_string, NULL);

    XiAnnotationPosition(annos[0], x0, y0, 8, 8);
    XiAnnotationPosition(annos[1], x1, y1, 8, 8);
    XiAnnotationPosition(annos[2], x2, y2, 8, 8);
    XiAnnotationPosition(annos[3], x3, y3, 8, 8);

    XtManageChildren(annos, 4);
}
```

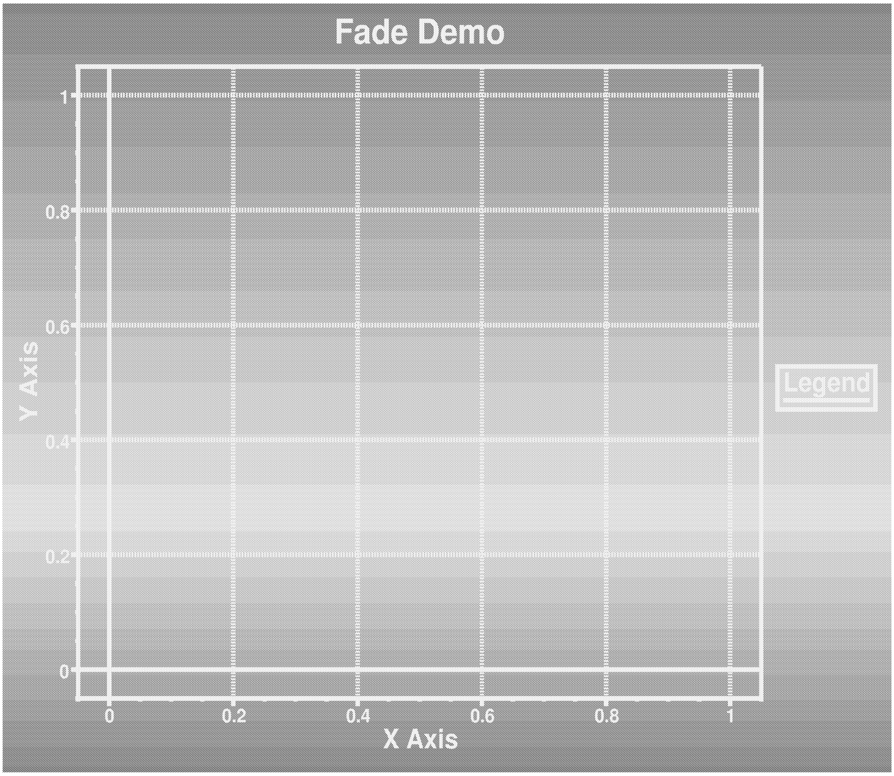
13.4 Updating Image Annotations

There is one final point to note in this chapter, and one final function to introduce. When using an ImagePlot widget, if you change the contents of a displayed bitmap, pixmap, or XImage without actually changing the value of the XmNbitmap, XmNpixmap, or XmNimage resources, you should call the function XiImagePlotUpdate() to inform the ImagePlot that its image has changed and that it needs to redraw it.

```
void XiImagePlotUpdate(Widget w);
```

`w` An ImagePlot widget which needs to be redisplayed. This function should be called after the contents (but not the value) of an image resource has been changed.

Figure 13-9. XiImagePlotUpdate()


Fade Demo
1
0.8
0.6
0.4
0.2
0
Y Axis
0
0.2
0.4
0.6
0.8
1
X Axis
Legend

14

The FadePlot Widget

The FadePlot widget is designed for presentation graphics. It adds visual interest to your graphs by drawing a background that "fades" smoothly from one color to another. This effect is most impressive in color, so the figure on the facing page is not representative of the capabilities of this widget.

FadePlot Synopsis

Class Name: XiFadePlot

Class Hierarchy: RectObj → XiPlot → XiFadePlot

Header File: *<Xi/FadePlot.h>*

Class Pointer: `xiFadePlotWidgetClass`

Constructor:

```
XiCreateFadePlot (Widget parent, String name,
                  ArgList args,
                  Cardinal num_args);
```

The FadePlot widget can be used as a background plot which appears within the plotting area, beneath the grid lines, or it can be used as a backdrop plot, and in this case it provides the background for the entire Plotter, as pictured on the facing page. The widget will automatically compute the colors of the "fade" it is to display, or it will display a list of colors that you specify. It will display the colors in any of four possible directions, and can interpolate them linearly or non-linearly.

The next section explains the FadePlot resources in detail, and the section after that shows an example of using the FadePlot and provides some hints for using it effectively.

14.1 FadePlot Resources

The FadePlot resources specify the colors that are to be displayed, how many colors there are, the direction they are displayed in, and how they are calculated. Other resources control how and whether those colors are printed in hardcopy. The resources are summarized in Table 14-1 and are explained in detail in the paragraphs that follow.

Table 14-1. FadePlot Resources

Name	Class	Type	Default	CSG	Pg
Colors					
XmNcolors	XmCColors	XmColorList	NULL	CSG	202
XmNstartColor	XmCStartColor	Pixel	XmNplotAreaColor[a]	CSG	202
XmNendColor	XmCEndColor	Pixel	XmNplotAreaColor[a]	CSG	202
XmNnumColors	XmCNumColors	Dimension	10	CSG	203
XmNfadeDirection	XmCFadeDirection	XiFadeDirection	top_to_bottom	CSG	203
Color Interpolation					
XmNuseHSL	XmCUseHSL	Boolean	FALSE	CSG	203
XmNinterpolateFunc	XmCInterpolateFunc	XiInterpolateFunc	NULL	CSG	203
Printing					
XmNdontPrint	XmCDontPrint	Boolean	FALSE	CSG	204
XmNpsNumColors	XmCNumColors	Dimension	100	CSG	204

[a] *Inherited from the Plotter XmNplotAreaColor resource.*

XmNcolors

An array of XmNnumColors Pixel values which you can set to specify an explicit list of colors for the FadePlot widget to display. If you do not specify a value for this resource, then the FadePlot will compute its own colors, based on the value of the XmNstartColor and XmNendColor resources. In this latter case, you can query this XmNcolors resource to obtain the array of computed colors the widget is displaying. When you set this resource, the widget does not make a copy of the array, so your array must remain valid for the lifetime of the widget. Similarly, when you query the resource, the returned array belongs to the widget, and should not be modified or freed.

XmNstartColor, XmNendColor

If the XmNcolors resource is not set, then the FadePlot will automatically compute its own colors to display by interpolating between the colors specified by these two resources.

XmNnumColors

If XmNcolors is set, then this resource must specify the number of Pixel entries in that array of colors. If XmNcolors is not set, then this resource specifies the number of colors that will be automatically computed for display. The number of colors specified by this resource include the two colors specified by the XmNstartColor and the XmNendColor resources, so, for example, if XmNnumColors is ten, then the FadePlot will interpolate eight new colors between the specified starting and ending colors.

XmNfadeDirection

The direction in which the colors will be displayed--the direction of the "fade". This resource is of type XiFadeDirection, and has one of four possible values (the meaning of which is self-explanatory): XiFADE_TOP_TO_BOTTOM, XiFADE_BOTTOM_TO_TOP, XiFADE_LEFT_TO_RIGHT, and XiFADE_RIGHT_TO_LEFT. The FadePlot registers a resource converter for this data type, so you can set this resource from a resource file. The converter recognizes the type names as you type them in C code, and it also allows you to omit the Xi and FADE_ prefixes. Further, it is case insensitive, so you can specify values like "XiFADE_BOTTOM_TO_TOP", "FADE_LEFT_TO_RIGHT", and even "top_to_bottom".

XmNuseHSL

This resource effects the way in which the FadePlot interpolates between XmNstartColor and XmNendColor. If False, then the FadePlot simply uses the RGB color coordinates of each color, and performs an interpolation between them. If True, then the FadePlot first converts the starting and ending color to HSL coordinates, and performs an interpolation in this Hue-Saturation-Lightness color space. The results of this interpolation may vary slightly or significantly, depending on the starting and ending colors. Because colors in the HSL color space vary in a more predictable fashion, you may find it easier to achieve particular fade effects that you are interested in by setting the XmNuseHSL resource to True. Note that the HSL color space is sometimes called the HSB (Hue-Saturation-Brightness) color space, or the HVC (Hue-Value-Chroma) space.

XmNinterpolateFunc

This resource specifies a function to be used to perform the color interpolation. If no function is specified, then the FadePlot will perform a straightforward linear interpolation. If a function is specified, it should be of type XiInterpolateFunc--a function that takes two integer arguments *i*, and *n*, and returns a double value between 0.0 and 1.0. The function specified by this resource will be called repeatedly to perform the interpolation; it will be called with a fixed *n* argument--the number of colors to be computed, and will be called with values for Ii that range between 0 and *n*. A return value of 0.0 indicates that the XmNstartColor should be used, and a return value of 1.0 indicates that the XmNendColor should be used. A return value between 0 and 1 specifies a color proportionally between the start and end colors. Specifying NULL for this resource is equivalent to specifying the following default function:

```
static double interpolate(int i, int n)
{
    return (double)i/(double)n;
}
```

XmNdontPrint

Because color "fades" may not be particularly interesting when printed on a monochrome PostScript printer, this resource allows you to turn printing of the FadePlot off. If set to True, then the FadePlot will not appear when the Plotter widget is printed. This is particularly useful when using dark colors in the fade--they would come out dark gray or black on a monochrome printer, and would obscure any other plots that appear.

XmNpsNumColors

This resource specifies the number of colors to interpolate when printing the fade in hardcopy. Because colors are a shared resource in X, and because many X displays can only display a limited number of distinct colors at one time, you may have to set the XmNnumColors to a smaller value than you'd like. Because these constraints do not exist in PostScript, it is possible (if your printer has sufficient pixel and color resolution) to achieve much smoother fades in hardcopy--note that the default value for this resource is 100, while the default for the XmNnumColors resource is only 10. If you set XmNpsNumColors to 0, then the XmNnumColors resource will be used when generating hardcopy instead. Note that this resource is not used if you have set the XmNcolors resource to specify an explicit array of colors to display in the fade.

14.2 Using the FadePlot

The FadePlot is a comparatively simple Plot type.Example 14-1shows a simple program that creates a FadePlot and uses it as the Plotter background. Example 14-0 is the resource file that was used with this program, to create the figure that appeared at the start of this chapter. When studying Example 14-1, note especially how the Plotter XmNbackdrop resource is set to indicate that the FadePlot should be drawn as the Plotter background. Also note the use of the XmNinterpolateFunc resource.

Example 14-1. Using the FadePlot widget

```
#include <X11/Intrinsic.h>
#include <Xi/Plotter.h>
#include <Xi/FadePlot.h>

static double interpolate(int i, int n)
{
    int t = n*2/3;
    if (i <= t) return (double)i/(double)t;
    else return (double)(n-i)/(double)(n-t);
}

void main(int argc, char **argv)
{
    XtAppContext app;
    Widget toplevel, plotter, fade;

    toplevel = XtVaAppInitialize(&app, "Demo", NULL, 0,&argc, argv,
    NULL,NULL);

    /* Create the plotter */
    plotter =XtVaCreateManagedWidget("plotter",xiPlotterWidgetClass,
      XmNtitle, "Fade Demo",
      toplevel,
      NULL);

    /* set up the Axis titles */
    XtVaSetValues(XiPlotterXAxis(plotter), XmNtitle, "X Axis",NULL);
    XtVaSetValues(XiPlotterYAxis(plotter), XmNtitle, "Y Axis",NULL);

    /* Create the FadePlot */
    fade = XtVaCreateManagedWidget("fade", xiFadePlotWidgetClass,
    plotter,
   XmNinterpolateFunc, interpolate,
   NULL);

    /* And make it the backdrop plot */
    XtVaSetValues(plotter, XmNbackdrop, fade, NULL);

    XtRealizeWidget(toplevel);
    XtAppMainLoop(app);
}
```

Example 14-2. FadePlot resources in a resource file

```
!! Use 50 colors interpolated between maroon2 and pink
*fade.startColor: maroon2
*fade.endColor: pink
*fade.numColors: 50

!! Draw the colors top-to-bottom (the default)
*fade.fadeDirection: top_to_bottom

!! Interpolate colors in the RGB color space, not the HSL color
space
*fade.useHSL: False

!! When printing, use 150 colors instead of 50.
*fade.psNumColors: 150

!! Set up other stuff in the plotter so it is visible
!! against a dark colored background.
*plotter.foreground: yellow
*plotter*LineWidth: 3
*plotter.legendBackground: lightblue
```

14.2.1 The FadePlot as Backdrop and Background

The FadePlot widget is generally most effective when used as a backdrop. As shown in Example 14-1, you can make it into the backdrop by setting the Plotter XmNbackdrop resource to point to the FadePlot. What this does is to tell the Plotter that your FadePlot needs to be drawn before anything else in the Plotter is drawn, and that it needs to be drawn as large as the entire Plotter widget, instead of simply being drawn within the plotting region.

You can also use the FadePlot as a background plot. A background plot is drawn within the plotting area, as other plots are, but it is drawn before any other plots, and is also drawn before any grid lines are. This effectively makes it the background of the plotting area. To display a FadePlot as a background plot, simply set the XmNbackgroundPlot constraint resource to True on the FadePlot. (Note this carefully: the XmNbackdrop resource is set on the Plotter widget, but the XmNbackgroundPlot resource is a constraint that is set on the FadePlot itself.) It is possible to have more than one background plot drawn at a time, but if you do this, be sure to set the drawing order of the plots carefully, as a FadePlot fills the entire plotting area, and will obliterate any background plots drawn before it. For this same reason you will never want to use a FadePlot unless you have made it a backdrop or a background plot--if drawn as a regular plot, it will erase the grid lines and any plots that are drawn before it is.

14.2.2 Non-Linear Color Interpolation

As we've seen the XmNinterpolateFunc resource can be used to produce fade colors using a non-linear interpolation. Example 14-3 shows some sample interpolation functions that you might use.

Example 14-3. Example non-linear interpolation functions

```
/*  Default function -- linear interpolation */
double interpolate1(int i, int n)
{
    return (double)i/(double)n;
}

/* Interpolate based on a non-linear function: y = x*x  */
double interpolate2(int i, int n)
{
    return (double)(i*i)/(double)(n*n);
}

/* Interpolate backwards -- same as changing the XmNfadeDirection
resource */
double interpolate3(int i, int n)
{
    return (double)(n-i)/(double)n;
}

/* Cycle through the colors twice */
double interpolate4(int i, int n)
{
    return (double)((i*2)%n) / (double)n;
}

/* interpolate forwards 2/3 of the screen, then backwards for 1/3 */
static double interpolate5(int i, int n)
{
    int t = n*2/3;
    if (i <= t) return (double)i/(double)t;
    else return (double)(n-i)/(double)(n-t);
}
```

14.2.3 Note on Color Usage

There are several interesting points about color usage that you should be aware of when using the FadePlot.

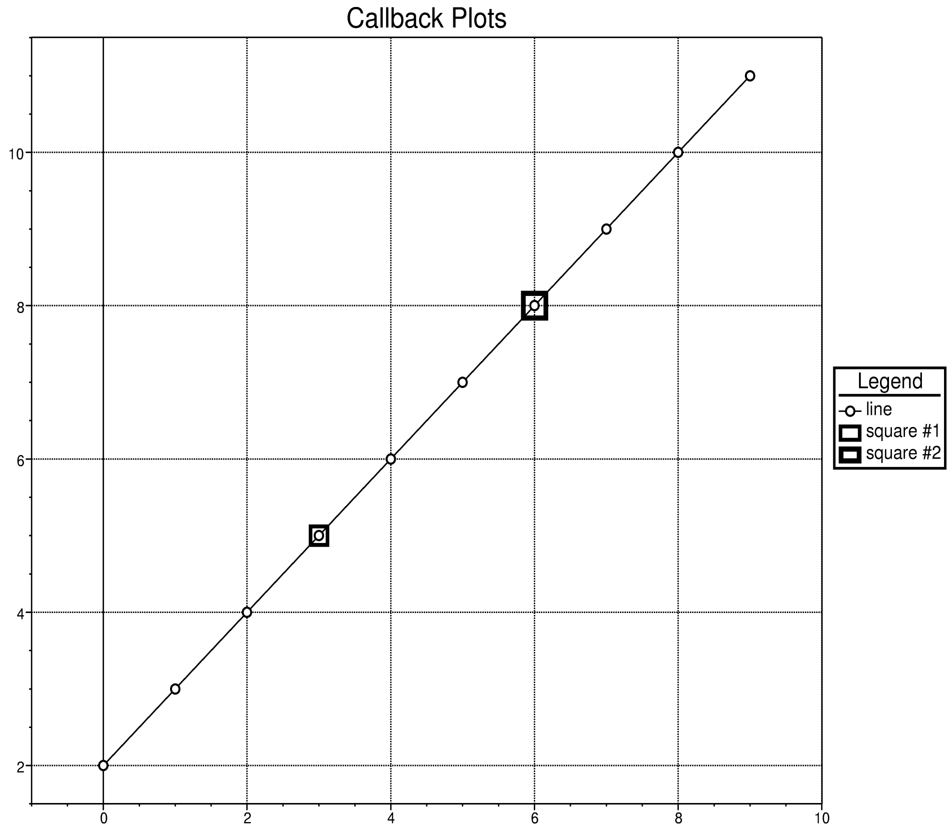
First, note that because colors are a shared resource in X, the FadePlot may not always be able to allocate the number of colors you specify in the XmNnumColors resource. If this happens, the FadePlot will print a warning message to the standard error stream, and will reuse the last color it was able to allocate for any remaining colors.

On the other hand, however, you can sometimes set the XmNnumColors resource to be larger than the number of colors available in the X colormap. This happens when the XmNstartColor and XmNendColor are similar enough that the X server can only display a small number of distinct colors between them. For example, many X servers can only display about 40 distinct colors between "sea green" and "medium sea green", so if you choose these colors as your starting and ending points, then you can set XmNnumColors as large as you want, and the FadePlot will never actually allocate more than 40 colors. A similar point is true of PostScript hardcopy--you can set XmNpsNumColors as large as you want, but the actual number of distinct colors that you see in the resulting output will depend on the color resolution of the output device.

Throughout this chapter, we've been assuming that you'll set the XmNstartColor and XmNendColor resources and have the FadePlot automatically compute the colors in between. You can, of course, allocate your own colors, and specify them directly on the XmNcolors resource. There are three situations in which you might like to do this. The first is to achieve special effects--with the XmNcolors resource you can display whatever colors you want, in whatever order you want. This gives you even more flexibility than is possible with your own XmNinterpolateFunc.

The second reason to use the XmNcolors resource is if you want to animate the colors of the background. If you allocate your colors in private read-write color cells, and then use these colors in the XmNcolors array, you can use XStoreColors() to change the RGB color associated with each color cell in the array. Doing this changes the displayed colors instantly. So, for example, by rotating a set of smoothly interpolated colors through an array of private color cells, you can achieve the effect of a moving background.

The third reason to use the XmNcolors resource is if you want both to have a FadePlot background and allow the user to use the Plotter's crosshair and "rubberbanding" capabilities. The plotter draws crosshairs and boxes regions swept out by the mouse by using XOR drawing. This means that the color of the crosshair or the box is dependent on the background color. With a FadePlot in the background, the color is never constant, and the crosshairs and boxes are usually almost invisible and impossible to use. A workaround to this problem is to use XAllocColorCells() to allocate your colors and to allocate a plane mask for those colors. Suppose you needed 25 distinct colors for your "fade". Your call to XAllocColorCells() would allocate 25 color cells, and would then also allocate another 25 cells each of which has an additional bit set in it. For example, if it allocated cells 10 through 34, it might also allocate cells 138 through 162. Then you can use the returned 25 cells for your interpolated fade colors, and set each of the additional 25 cells to the fixed color that you want the crosshairs and boxes to appear in. Finally, you use specify the returned "plane mask" on the XmNcrosshairColor and related resources, and the Boolean algebra of XOR will ensure that the proper color is displayed, regardless of the FadePlot background colors.


Callback Plots
10
8
6
4
2
0
2
4
6
8
10
Legend
line
square #1
square #2

15

Drawing Your Own: the CallbackPlot

The CallbackPlot widget is a very simple widget class that displays no data of its own, but instead provides callback hooks to enable you to display data in any format you choose.

CallbackPlot Synopsis

Class Name: XiCallbackPlotPlot

Class Hierarchy: RectObj → XiPlot → XiCallbackPlot

Header File: <*Xi/CallbackPlot.h*>

Class Pointer: `xiCallbackWidgetClass`

Constructor:

```
XiCreateCallbackPlot (Widget parent,
                      String name, ArgList
                      args, Cardinal num_args);
```

When you create a CallbackPlot, it will do nothing on its own. If you define an appropriate callback, however, you can make it do any kind of drawing you want. By defining another callback, you can have it draw a shadow for your custom plot type, or by defining a third callback, you can tell the CallbackPlot how to draw your custom plot type in PostScript.

Note that in order to draw your own custom plots on the screen with the CallbackPlot widget, you'll need to be familiar with Xlib, and to transfer those custom plots to hardcopy, you'll need to know how to write simple PostScript code.

The following section explains the CallbackPlot resources. It is followed by an example of creating a CallbackPlot and defining callbacks that display a custom plot type.

15.1 CallbackPlot Resources

The CallbackPlot has 10 callback resources, 10 "data" resources that are untyped and are for your own use, and a single "control resource" that you can use to tell the CallbackPlot to redraw itself. These resources are summarized in Table 15-1 and are explained in detail in the subsections that follow.

Table 15-1. CallbackPlot Resources

Name	Class	Type	Default	CSG	Pg
Untyped Data					
XmNdata0	XmCData	XiUntyped	NULL	CSG	213
XmNdata1	XmCData	XiUntyped	NULL	CSG	213
XmNdata2	XmCData	XiUntyped	NULL	CSG	213
XmNdata3	XmCData	XiUntyped	NULL	CSG	213
XmNdata4	XmCData	XiUntyped	NULL	CSG	213
XmNdata5	XmCData	XiUntyped	NULL	CSG	213
XmNdata6	XmCData	XiUntyped	NULL	CSG	213
XmNdata7	XmCData	XiUntyped	NULL	CSG	213
XmNdata8	XmCData	XiUntyped	NULL	CSG	213
XmNdata9	XmCData	XiUntyped	NULL	CSG	213
Callbacks					
XmNchangeCallback	XmCCallback	Callback	NULL	C	217
XmNcreateCallback	XmCCallback	Callback	NULL	C	217
XmNdrawCallback	XmCCallback	Callback	NULL	C	217
XmNdrawPSCallback	XmCCallback	Callback	NULL	C	217
XmNfreeCallback	XmCCallback	Callback	NULL	C	218
XmNlegendCallback	XmCCallback	Callback	NULL	C	218
XmNlegendPSCallback	XmCCallback	Callback	NULL	C	219
XmNresizeCallback	XmCCallback	Callback	NULL	C	219
XmNshadowCallback	XmCCallback	Callback	NULL	C	219
XmNshadowPSCallback	XmCCallback	Callback	NULL	C	219
Control Resources					
XmNrecalc	XmCRecalc	Boolean	FALSE	S	220
XmNredraw	XmCRedraw	Boolean	FALSE	S	220
XmNxMax	XmCXMax	double	-DBL_MAX	CSG	220
XmNxMin	XmCXMin	double	DBL_MAX	CSG	220
XmNyMax	XmCYMax	double	-DBL_MAX	CSG	220
XmNyMin	XmCYMin	double	DBL_MAX	CSG	220

15.1.1 Untyped Data Resources

The CallbackPlot has an unusual set of resources with no fixed data type, and no set purpose. This is fitting, though, since the CallbackPlot itself has no fixed purpose, either. When you write the callback procedures that will make a CallbackPlot widget display some custom plot type, you can use the XmNdata0 through XmNdata9 resources for any purpose you want. Each callback procedure will be passed *call_data* that allows it to easily get and set the values of these resources. You can use them as scratch variables, as flags for communication between the various callback routines.

You can also use these data resources more traditionally, as actual resources, so that users of your custom plot type (i.e. you or other programmers) can specify parameters that control the plots appearance or behavior. You might decide that the XmNdata0 resource will specify the foreground color for your custom plot, that the XmNdata1 resource will specify a XFontStruct*, for the plot, and that XmNdata2 will specify the width, in pixels, of the lines to be drawn by the plot. Then, once you had written the appropriate callback procedures that used these resource values, you could create multiple instances of your custom plot type, which all used different colors, fonts, and line widths.

There is nothing tricky about setting the value of these untyped data resources. When you use argument lists, the XtSetArg() macro casts all resource values to an XtArgVal anyway, so in effect all resources are untyped. Similarly, if you specify resources in a call to XtVaCreateWidget(), all the resource values in the variable-length argument list are implicitly cast to integers, anyway. (Do note, though, that setting resources with XtSetValues() or XtVaSetValues() may only work if you've written a callback that notices and handles these changes.) You can always query the value of these resources with XtGetValues() or XtVaGetValues(), of course.

Note in Table 15-1 that the data resources all have representation type XmRXiUntyped. The CallbackPlot widget defines a String-to-XiUntyped resource converter, which means that you (or your application's users) can actually set these untyped resources from a resource file. The trick is that you must specify both resource value and the type that the resource is to be converted to. For the purposes of resource conversion, types are specified as the string values of their XtR representation types. For example, you could use the XiUntyped converter with resources like the following:

```
! foreground color
*cbplot1.data0: Pixel: red
! font
*cbplot1.data1: FontStruct: *-helvetica-medium-r-*-*-*-140-*
! line width
*cbplot1.data2: 4
```

Note the use of a colon to separate the data type from the data value in the example. Also note that in the last resource, we didn't specify a data type at all. If the XiUntyped converter can't find a type specification in the resource, it assumes that the value is an integer.

The paragraphs above explain how to set resource values from "the outside"--as a user of the custom plot type. But when you're writing the callbacks that define the widget's behavior, you'll be able to read and write these data values directly, as elements in an array. While access is more direct in this case, casting is not handled automatically. Each of the data values in the array is of type XtPointer, so you'll have to cast to and from this type when setting or reading the values of these variables. You'll see examples of this direct reading and setting of the untyped data variables in the example at the end of the chapter.

An important shortcoming of these untyped data resources is that they are declared to be XtPointer variables. On most architectures, this is not big enough to hold a double value. If you want to specify floating-point values with these resources, you'll either have to use float variables, or use pointers to double. If you use pointers, note that your callback procedures should either make a private copy of the double value, or you should be very careful to only specify the address of double variables that are stored in static memory, or allocated memory that won't be freed. If you pass the address of a double value that is an automatic variable (i.e. a variable on the stack) the value of the resource you set will probably be trashed as soon as the current procedure returns and a new procedure is called. The extended example that ends this chapter shows how you might use pointers to double variables as resource values.

The CallbackPlot provides ten of these untyped data resources, which is enough for simple plot types. More complicated plots may need more space than this, however, to use as scratch variables or to maintain its internal state. If this is the case for your custom plot type, simply define a data structure to contain all the extra fields you need, allocate one of these structures for each widget instance, and then use one of the existing data resources (XmNdata9, for example) to point to the allocated instance of this data structure. You can even handle the allocation and deallocation of this extra data structure from within the callbacks that you write for your custom plot type.

15.1.2 Callback Resources

The most significant resources of the CallbackPlot widget are, not surprisingly, its callbacks. The widget class defines ten callback lists, each of which provides a hook for a single aspect of plot behavior. Each of these callbacks are invoked from within the various class methods of the CallbackPlot widget, so what they do is give you the opportunity to insert arbitrary C code into each of the widget class methods--in effect, defining your own widget class.

Each of the callbacks of the CallbackPlot is passed a pointer to an XiCallbackPlotStruct as its *call_data* argument. This structure is shown in Figure 15-1; descriptions of the individual fields follow. Note that not all fields of this structure are defined for each of the callbacks; the descriptions of the individual callbacks below specify which fields are valid.

```
  typedef struct {
    Display *display;             /* The X display */
    Drawable drawable;           /* What to draw into; usually a window */
    GC gc;                        /* A gc that is all yours to use */
    Region region;                /* A clipping region for efficiency */
    FILE *file;                   /* The file to output PostScript to. */
    XRectangle rectangle;        /* drawing area in pixels or PS points */
    double xmin, xmax, ymin, ymax;/* the axis bounds */
    XtPointer *data;              /* address of XmNdata0 to XmNdata9 */
    XtPointer *olddata;          /* old datavalues for XmNchangeCallback*/
} XiCallbackPlotStruct;
```

Figure 15-1. XiCallbackPlotStruct

The fields of the XiCallbackPlotStruct are the following:

display

This field is just the Xlib Display pointer. You'll use it for any drawing you do.

drawable

This is the X window to draw into. Note that in the XiCallbackPlotStruct, this field is declared as type Drawable, rather than as type Window. This is because of double-buffering--when double-buffering is turned on, you may actually end up drawing into a Pixmap or a Multibuffer. A Drawable behaves almost just like a Window, and you never need to know what kind of Drawable you are drawing into. The only thing to look out for is that you cannot call XClearWindow() or XClearArea() on a Drawable--instead, you have to use XFillRectangle().

gc

This field is an Xlib GC--a graphics context--that the CallbackPlot automatically creates for your use. Each CallbackPlot widget gets its own private GC, which you can use it for absolutely anything you want. The CallbackPlot never sets any attributes in this GC--you set whatever attributes you need to whatever values you want. Note that the GC is automatically created when the widget is created and automatically destroyed when the widget is destroyed.

region

A Region is an opaque Xlib structure that defines an arbitrary region of the screen. It is used in Xt to specify the area of a window that needs to be redrawn when an Expose event (or a series of Expose events) occurs. If the region field is non-NULL, you can use it to help make redrawing your plot more efficient--use XRectInRegion() to find out if any part of your drawing needs to be redrawn. Or, use XClipBox() and XSetClipRectangles() to define a clipping region so that the X server doesn't do more drawing than is necessary.

file

This field is defined for the callbacks that allow you to generate PostScript hardcopy of your custom plot type. It is a FILE * to which you should output your PostScript commands.

rectangle

This field is an XRectangle structure, which contains x, y, width, and height fields. These four values define a rectangle within which you should do your drawing. The rectangle either gives the coordinates of the main plotting area of the Plotter widget (i.e. the area between the axes), or it gives the coordinates of a small rectangle in the legend into which your custom plot can draw an iconic representation of itself. For the X drawing callbacks, the rectangle will be expressed in X coordinates--i.e pixels. For the PostScript drawing callbacks, the coordinates of the rectangle will be expressed in the PostScript coordinate system. In any case, you should never draw anything that falls outside of this rectangle.

xmin, xmax, ymin, ymax

These four fields define the minimum and maximum values displayed on the x and y axes. You can use these values along with the rectangle field to scale your plot and position it appropriately in the plotting area.

data

This field gives the address of the XmNdata0 resource. Since the XmNdata0 through XmNdata9 resources are stored as an array of XtPointer, you can access the value of each of these untyped resources just by using this data field with array notation: data[0], data[9], etc. Note that you can both get *and* set the value of these resources in this way.

olddata

This field is the address of a copy of all the data resources. It is only used for the XmNchangeCallback, when the user has changed the value of a data resource by calling XtSetValues() or XtVaSetValues().

Having explained all of the fields in the XiCallbackPlotStruct, we can finally explain each of the callbacks themselves. The extended example at the end of this chapter shows how you might implement many of these callbacks.

XmNchangeCallback

The XmNchangeCallback is invoked when XtSetValues() or XtVaSetValues() is called on the widget, and one or more of the XmNdata*n* resource values has changed. The display, gc, data, and olddata fields of the XiCallbackPlotStruct are defined for this callback. The purpose of this callback is to allow you to make any necessary updates to the widget's internal state to respond to the changes. For example, if you are using XmNdata2 as the foreground color for your plot, then you might want to update your GC if this resource changes. You could tell whether this value has changed by comparing data[2] (the current value) to olddata[2] (the value before the call to XtSetValues()).

XmNcreateCallback

This callback is called when an instance of a CallbackPlot is first created, but after all of its resource values have been set to their initial values. Only the display, gc, and data fields of the XiCallbackPlotStruct valid for this callback. The purpose of the callback is to give you an opportunity to initialize any internal state of the widget. For example, you might set fields in the GC based on resource values, and you might allocate and initialize any additional data structures that the widget will need. This callback resource need not actually be specified when the widget is created. It is legal to create the CallbackPlot widget, and then specify the callback with XtAddCallback(), as long as this is done before XtSetValues() or XtManageChild() is called for the widget. In any case, if the XmNcreateCallback is invoked, it will be invoked before the XmNresizeCallback, and before the XmNchangeCallback, and it will never be invoked more than once for any given widget.

XmNdrawCallback

This callback is invoked whenever any part of the plotting area of the Plotter widget needs to be redrawn. The display, drawable, region, gc, rectangle, xmin, xmax, ymin, ymax and data fields are all valid. You should use this callback to make whatever Xlib calls are necessary to draw your custom plot. Be sure, however, that you do not draw anything outside of the region specified by the rectangle field. If the region field is not NULL, then you can use it to check whether you actually have to redraw your plot. Note that if your plot type has a shadow, you should not draw that here--instead, use the XmNshadowCallback.

XmNdrawPSCallback

This callback is invoked when a Plotter widget that is displaying your CallbackPlot is printed. The file, rectangle, xmin, xmax, ymin, ymax, and data fields are valid for this callback. This callback should output the PostScript commands to draw itself to the file specified by file. The PostScript commands you output will automatically be surrounded by a gsave/grestore pair, so that nothing you do will have any permanent effect on the PostScript graphics state. The xmin, xmax, ymin, and ymax specify the bounds of the axes, as usual, but note that the rectangle structure contains the bounds of the plotting area in PostScript coordinates, rather than pixels. One important thing to note is that in PostScript, the Y coordinate increases as you go *up* the page, rather than as you go *down* the screen, as in X. Also note that the values in the rectangle structure will be much larger than they are when you draw your custom plot type in X--in order to take advantage of the high-resolution of PostScript, the Plotter widget sets up its coordinate system so that 16 PostScript units take up the same size on the page as 1 pixel takes up on the screen. This is quite important to remember. It means, for example, that if you want to draw a line that is 2 pixels wide, you'll set the PostScript line width with a command like:

```
fprintf(call_data->file, "32 setlinewidth\en");
```

Finally, note that there is no resize callback called before a CallbackPlot is printed. This means that your XmNdrawPSCallback will have to be responsible for doing any necessary scaling computation as well as doing the drawing. You may find that you'll write your XmNdrawPSCallback by combining your XmNresizeCallback with your XmNdrawCallback, and converting your Xlib calls to PostScript output.

XmNfreeCallback

This callback is called when a CallbackPlot is destroyed. Only the display and data fields of the XiCallbackPlotStruct are valid. You might use this callback to deallocate any memory or other resources that you have allocated for the widget. Note that you do not have to deallocate the GC that the CallbackPlot provides--this GC is freed automatically by the widget. Also note that this callback is different from the XmNdestroyCallback that the CallbackPlot widget inherits from the RectObj class.

XmNlegendCallback

This callback is called when the Plotter legend needs to be redraw, if your CallbackPlot has an entry in the legend. The callback's purpose is to allow you to draw a small iconic representation of the plot within the legend. The display, drawable, gc, rectangle, and data fields are all valid for this callback. The rectangle structure will describe a fairly small rectangle within the legend. You should be sure not to draw anything outside of this rectangle, but also be sure to use all of the space provided for your plot icon. Your icon should use the same colors, line patterns, fill patterns, or special marks that your plot uses, so users will be able to recognize the plot that it refers to.

XmNlegendPSCallback

This callback is called when a Plotter containing a CallbackPlot is printed, and that CallbackPlot has an entry in the legend. The file, rectangle, and data fields are valid for this callback. In this callback, you should output the necessary PostScript commands to file to draw an icon for your plot within the rectangle specified by rectangle. Remember that the rectangle is in PostScript coordinates, where 16 units equal one pixel, and where the direction of the Y coordinate is reversed. As with the XmNdrawPSCallback, your output will automatically be encapsulated within an gsave\grestore pair.

XmNresizeCallback

This callback is invoked whenever the size of the Plotter's plotting area changes or when any of the axis bounds change. It is also invoked when the CallbackPlot is first created, before the XmNdrawCallback is ever invoked. The display, gc, rectangle, xmin, xmax, ymin, ymax, and data fields are valid. The purpose of this callback is to do any scaling or other pre-computation that you have to do when the widget changes size. For example, you might use this callback to compute the pixel coordinates that correspond to a given floating-point data value. Then in the XmNdrawCallback, you'll only have to draw your custom plot at these pixel coordinates, rather than having to do the computation. The example at the end of this chapter shows how you can perform this kind of scaling. You might also find the functions XiConvertWorldToPixel() and XiConvertPixelToWorld() useful in this callback. These functions are documented in Section 6.4, *World <-> Pixel Coordinate Conversion*.

XmNshadowCallback

The XmNshadowCallback is invoked when the plotting area is redrawn in order to give you the opportunity to draw a shadow beneath your plot. The valid fields and their usage is identical to the XmNdrawCallback. The reason shadows must be drawn separately, and not as part of the XmNdrawCallback is that plots may overlap, but a plot shadow should not overlap another plot. Thus all plot shadows must be drawn before any plots are drawn. To draw a shadow you generally just draw your plot as you normally would, but draw it in gray, or with a stipple, and draw it offset two or three pixels in each dimension from where you'd normally draw it.

XmNshadowPSCallback

This procedure is called when a CallbackPlot is printed, to give you the opportunity to draw a shadow beneath your plot. The valid fields of the XiCallbackPlotStruct, and their interpretation are exactly the same as for the XmNdrawPSCallback. See XmNshadowCallback for more information on how and why to draw a shadow.

15.1.3 Control Resources

Most ordinary widget resources are used to specify a parameter that modifies the widget's appearance or behavior. The CallbackPlot is not an ordinary widget, however, and the resources in this section are used to *send a command* to the widget.

XmNrecalc

If you set this resource to True, the XmNresizeCallback and the XmNdrawCallback will all called in order to recompute and redraw your custom plot. If you also set any of the data resources in the same call to XtSetValues() or XtVaSetValues(), then the XmNchangeCallback will be invoked before either the above two. You would set this resource, for example, when you've changed the data that is to be displayed, or when you've changed a resource that will effect how that data is scaled. After you set this resource to True, its value is always automatically reset to False.

XmNredraw

This resource is similar to the XmNrecalc resource. When you set it to True, it causes the XmNdrawCallback() to be invoked, and your custom plot will be redrawn. If you also set any of the data resources in the same call to XtSetValues() or XtVaSetValues(), then the XmNchangeCallback will be invoked before the XmNdrawCallback is. You would set this resource, for example, when changing a data resource that specifies the color of the plot. This way, the XmNdrawCallback would be invoked to redraw the plot in the new color.

XmNxMax, XmNxMin, XmNyMax, XmNyMin

These four resources are variables of type double, and they specify the bounding box, in axis coordinates, for your custom plot. This bounding box information is necessary in order for Plotter autoscaling to work correctly. If you'll be using autoscaling, set these resources to the largest and smallest X and Y values that your plot displays. Do not confuse these with the xmax, xmin, ymax and ymin fields that appear in the XiCallbackPlotStruct structure. Those fields specify the bounds of the X and Y axes, which may be larger, or even smaller than the bounds of any particular plot. Since the value of these resources is generally computed based on the data to be plotted, you'll most often want to set these resources from inside of the XmNcreateCallback or XmNchangeCallback. Setting these bounding box resources will not cause the XmNchangeCallback to be invoked (or re-invoked). Generally when these resources are set in response to a data change, the XmNrecalc resource will also have been set, and in the ensuing recalculation, the new bounding box will be taken into account for autoscaling. If you ever set these resources from *outside* the widget callbacks, note that the changes won't take effect until you also set the XmNrecalc resource.

15.2 CallbackPlot Example

Example 15-1 is a complete example of using the CallbackPlot to define a simple custom plot--one that draws a square around a given point, as a way of highlighting that point. This program was used to generate the figure that began this chapter.Example 15-1 shows a resource file that you could run this program with. When studying this example, note how the procedure used for the XmNcreateCallback and the XmNchangeCallback updates GC fields. Note how the XmNresizeCallback scales axis coordinates to pixel coordinates. And given the work done in those callbacks, notice how simple the XmNdrawCallback is.

Example 15-1. Using the CallbackPlot widget

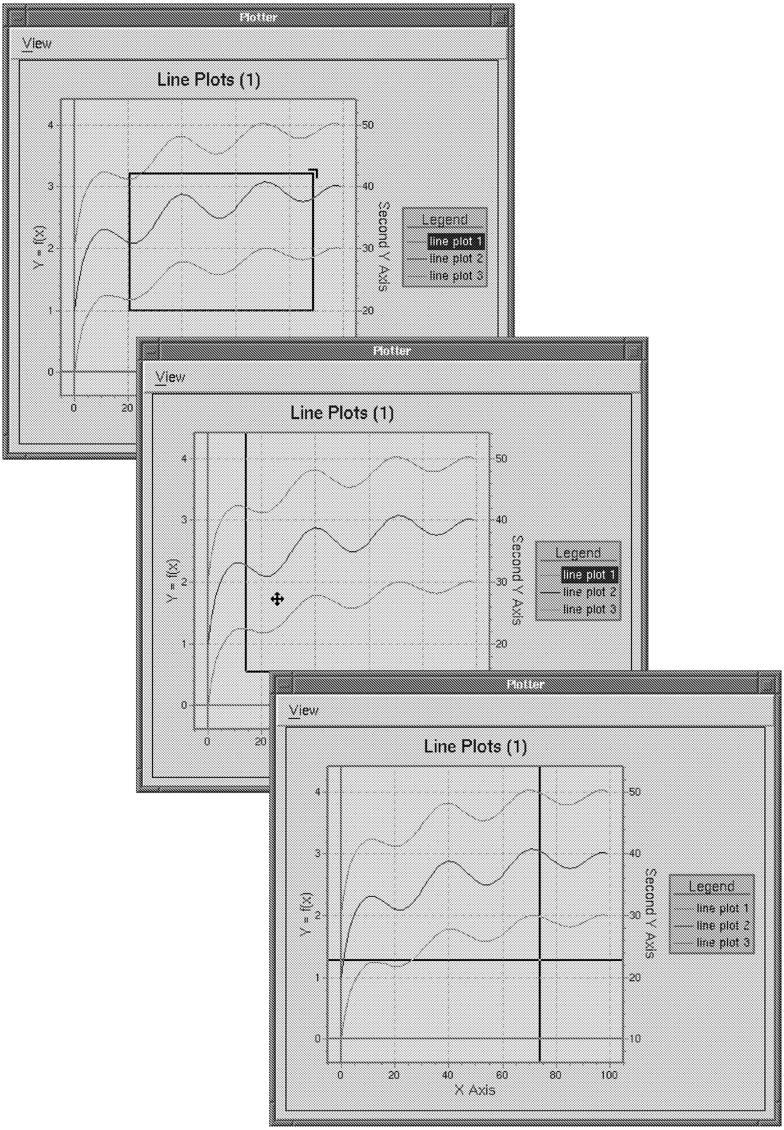
```
#include <stdio.h>
#include <X11/Intrinsic.h>
#include <Xi/Plotter.h>
#include <Xi/LinePlot.h>
#include <Xi/CallbackPlot.h>

/*
 * The following are a set of callbacks that will make CallbackPlot
 * draw a rectangle around a specified point. The callbacks expect
 * and use the data resources as follows:
 *   data0: pointer to a double--the x coordinate
 *   data1: pointer to a double--the y coordinate
 * Those two resources can't be set from a resource file, and the
 * values must be stored in allocated or static memory.
 *
 * Other data values:
 *   data2: a Pixel--the color of the box
 *   data3: an int--the length of each edge of the box
 *   data4: an int--the thickness of the lines of the box.
 *
 * Internally, the widget uses the following:
 *   data5: the x pixel coordinate of the upper left.
 *   data6: the y pixel coordinate of the upper left.
 */

/* This procedure used for both create and change callback */
static void create_and_change(Widget w, XtPointer tag, XtPointer
call_data)
{
    XiCallbackPlotStruct *data = (XiCallbackPlotStruct *)call_data;

    XSetForeground(data->display, data->gc, (Pixel)data->data[2]);
    XSetLineAttributes(data->display, data->gc, (int)data->data[4],
       LineSolid, CapButt, JoinRound);
}

/*
 * The resize callback. Note how we convert axis coordinates to
 * pixel coordinates, to save computation in the draw callback
 */
```

```
static void resize(Widget w, XtPointer tag, XtPointer call_data)
{
    XiCallbackPlotStruct *data = (XiCallbackPlotStruct *)call_data;
    double x, y;
    int x0, y0;
    int box_size;

    /*
     * Figure out the pixel coordinates of the center of the box
     * We do this coordinate conversion explicitly here, to show
     * how it is done. You could also call XiConvertWorldToPixel().
     */
    x = *(double *)data->data[0];
    y = *(double *)data->data[1];

    x0 = (x - data->xmin)/(data->xmax - data->xmin) * data->rectan-
gle.width
+ data->rectangle.x;
    y0 = (data->ymax - y)/(data->ymax - data->ymin) * data->rectan-
gle.height
+ data->rectangle.y;

    /* Convert to upper left of box */
    box_size = (int)data->data[3];
    x0 -= box_size/2;
    y0 -= box_size/2;

    /* store this away for later use */
    (int)data->data[5] = x0;
    (int)data->data[6] = y0;
}

/* Draw the plot */
static void draw(Widget w, XtPointer tag, XtPointer call_data)
{
    XiCallbackPlotStruct *data = (XiCallbackPlotStruct *)call_data;

    XDrawRectangle(data->display, data->drawable, data->gc,
   (int)data->data[5], (int)data->data[6],
   (int)data->data[3], (int)data->data[3]);
}

/* Draw the plot icon in the legend */
static void legend(Widget w, XtPointer tag, XtPointer call_data)
{
    XiCallbackPlotStruct *data = (XiCallbackPlotStruct *)call_data;
    int linewidth = (int) data->data[4];

    XDrawRectangle(data->display, data->drawable, data->gc,
   data->rectangle.x + linewidth/2,
   data->rectangle.y + linewidth/2,
   data->rectangle.width - linewidth,
   data->rectangle.height - linewidth);
}

/*
 * Draw the plot in Postscript. Note how we scale the points
 * again. And use an upside-down Y coordinate.
 */
static void drawps(Widget w, XtPointer tag, XtPointer call_data)
{
    XiCallbackPlotStruct *data = (XiCallbackPlotStruct *)call_data;
    double x, y;
    int x0, y0, width, height;
    int box_size, line_width;

    /*
     * For PostScript, we've got to do all the scaling in the same
     * way we did it in the resize callback. Except note that the
     * Y units increase up the page rather than down.
    * And don't forget, in the units we use, 1 pixel is 16 PS units,
     * so a 2 pixel wide line should have a PS width of 32.
     */
    x = *(double *)data->data[0];
    y = *(double *)data->data[1];

    x0 = (x - data->xmin)/(data->xmax - data->xmin) * data->rectan-
gle.width
+ data->rectangle.x;
    y0 = (y - data->ymin)/(data->ymax - data->ymin) * data->rectan-
gle.height
+ data->rectangle.y;

    box_size = (int)data->data[3] * 16;
    line_width = (int)data->data[4] * 16;

    x0 -= box_size/2;
    y0 -= box_size/2;

    fprintf(data->file, "%d setlinewidth\en %d %d %d %d rect-
stroke\en",
    line_width, x0, y0, box_size, box_size);
}

/* Draw the plot icon for the legend, in Postscript */
static void legendps(Widget w, XtPointer tag, XtPointer call_data)
{
    XiCallbackPlotStruct *data = (XiCallbackPlotStruct *)call_data;
    int line_width = (int)data->data[4] * 16;

    fprintf(data->file, "%d setlinewidth\en %d %d %d %d rect-
stroke\en",
    line_width,
    data->rectangle.x + line_width/2,
    data->rectangle.y + line_width/2,
    data->rectangle.width - line_width,
    data->rectangle.height - line_width);
}

/*
 * A convenience function for creating the widget and registering
callbacks
 */
Widget CreateSquareWidget(Widget parent, String name,
  ArgList args, Cardinal num_args)
{
    Widget w;

    w = XtCreateWidget(name, xiCallbackPlotWidgetClass, parent,
       args, num_args);

    XtAddCallback(w, XmNcreateCallback, create_and_change, NULL);
    XtAddCallback(w, XmNchangeCallback, create_and_change, NULL);
    XtAddCallback(w, XmNresizeCallback, resize, NULL);
    XtAddCallback(w, XmNdrawCallback, draw, NULL);
    XtAddCallback(w, XmNlegendCallback, legend, NULL);
    XtAddCallback(w, XmNdrawPSCallback, drawps, NULL);
    XtAddCallback(w, XmNlegendPSCallback, legendps, NULL);

    return w;
}

void main(int argc, char **argv)
{
    XtAppContext app;
    Widget toplevel, plotter, cbplot1, cbplot2, lineplot;
    double m, c;
    double x1, y1, x2, y2;
    Arg al[10];
    int ac;

    toplevel = XtVaAppInitialize(&app, "hello", NULL, 0,
 &argc, argv, NULL,
 NULL);

    plotter = XtVaCreateManagedWidget("plotter", xiPlotterWidget
              Class,
      toplevel,
      XmNwidth, 500,
      XmNheight, 400,
      XmNtitle, "Callback Plots",
      NULL);

    XiAxisAutoscale(XiPlotterXAxis(plotter));
    XiAxisAutoscale(XiPlotterYAxis(plotter));

    m = 1.0;
    c = 2.0;
    lineplot = XtVaCreateManagedWidget("lineplot", xiLinePlotWidget
              Class,
       plotter,
       XmNyValueMultiplier, &m,
       XmNyValueConstant, &c,
       XmNnumValues, 10,
       XmNmarkPoints, True,
       XmNlegendName, "line",
       NULL);

    /*
     * Remember: given the way we've defined the callbacks,
     * these data0 and data1 resources are not copied, and
     * the double that they point to must remain valid for
     * the life of the widgets
     */
```

```
        x1 = 3.0; y1 = 5.0;
        ac = 0;
        XtSetArg(al[ac], XmNdata0, &x1); ac++;
        XtSetArg(al[ac], XmNdata1, &y1); ac++;
        XtSetArg(al[ac], XmNlegendName, "square #1"); ac++;
        cbplot1 = CreateSquareWidget(plotter, "cb1", al, ac);
        XtManageChild(cbplot1);

        x2 = 6.0; y2 = 8.0;
        ac = 0;
        XtSetArg(al[ac], XmNdata0, &x2); ac++;
        XtSetArg(al[ac], XmNdata1, &y2); ac++;
        XtSetArg(al[ac], XmNlegendName, "square #2"); ac++;
        cbplot2 = CreateSquareWidget(plotter, "cb2", al, ac);
        XtManageChild(cbplot2);

        XtRealizeWidget(toplevel);

        /* print out the plotter to test the PostScript callbacks */
        XiPlotterPrint(plotter, "testcallback.ps", 0, 0, 1.0, False);

        XtAppMainLoop(app);
    }
```

Example 15-2. Resources for the CallbackPlot Example

```
    *cb1.data2: Pixel: red
    *cb1.data3: Int: 15
    *cb1.data4: 3

    *cb2.data2: Pixel: navy
    *cb2.data3: 20
    *cb2.data4: 4
```


Plotter
View
Line Plots (1)
Y = f(x)
Second Y Axis
Legend
line plot 1
line plot 2
line plot 3
0
1
2
3
4
10
20
30
40
50
0
20
40
60
80
100
X Axis

16

User Interaction with the Plotter

The uses we've seen so far for the Plotter widget have all been static displays of data. But the Plotter is not an output-only widget--it also allows user input. The user can interact with the Plotter in six different ways:

- Clicking ("selecting") or double-clicking ("activating") items in the Plotter legend.
- Clicking or double-clicking on a point in the plotting area.
- Dragging the mouse with no buttons down, in order to display crosshairs in the plotting area.
- Dragging the mouse in the with the Ctrl key down, in order to drag out a rectangle and select a region of the plotting area.
- Dragging the mouse in the with the Meta key down, to move a rectangle around the screen, as a way of specifying how the plotting area should be scrolled.
- Clicking or double-clicking on the Plotter title, or on any of the axes.

When the user interacts with the Plotter in any of these six ways, the Plotter interacts with your application by invoking any callback procedures you have registered on any of its eleven callback lists. In addition to these callback lists, a number of other Plotter resources control whether and how the user can interact with the plotter. The following sections explain each of the six types of interaction, and explain the resources and callbacks associated with each.

You can also customize the mouse button and keyboard bindings used for the various types of user interaction by changing the default translation table. This chapter concludes with an explanation of the Plotter widget's action procedures, and the default Plotter translation table that binds user events to those actions.

16.1 Selecting Items in the Legend

The legend displayed by the Plotter behaves somewhat like a list widget: users can select individual plots displayed in the plotter by clicking, shift-clicking or double-clicking on them. In an application, for example, a user might select a plot by clicking on its key in the legend and then select a "Delete Plot" menu item. Or, a user might double-click on a plot in the legend as a shortcut to pop up a "Edit Plot Attributes" dialog box.

When the user selects an item in the legend with a single click, the selection is reported by the XmNselectCallback. When the user "activates" an item by double-clicking on it, the activation is reported by the XmNactivateCallback. Both these callbacks are passed a pointer to the following structure as their *call_data*.

```
/* callback reasons for the select and activate callbacks */
#define XmCR_SELECT        7 /* selection state changed in legend */
#define XmCR_ACTIVATE       8 /* double-click in legend */

typedef struct {
    int reason;        /* the type of the callback */
    XEvent *event;     /* the event that triggered it */
    Boolean selected;  /* True if an item was selected; False if de-
selected */
    Widget item;       /* the item that was selected, if any */
   Widget *selected_plots;     /* a complete list of selected plots
*/
    Cardinal num_selected_plots; /* # of elements in the list */
} XiSelectCallbackStruct, XiActivateCallbackStruct;
```

The following paragraphs describe the two callbacks and the meaning of the fields in the above structure.

XmNselectCallback

The list of callback procedures to be called when the user selects or deselects an item in the legend. (Note that if the user clicks on an already selected item, there is no change to the selection state of the widget, and this callback list is not invoked.) The call_data argument is of type XiSelectCallbackStruct* and points to a structure that contains a pointer to the selected or deselected plot, as well as an array of all currently selected plots.

The selected field of this structure specifies whether an item was selected or deselected. If it is True, then the item field specifies the item that was just selected. If selected is False, then item either specifies a single item that was deselected (this can only be done by using a shift-click to toggle an item) or item is NULL, which indicates that all the selected items were de-selected because the user clicked in the title of the legend.

The selected_plots field of this callback structure is an array of all the selected children in the widget, and num_selected_plots is the number of elements in that array. If the user selected a widget with a click, then there will always be just one element in this array. If the user selected a widget with a shift-click, then there may be multiple elements in the array. The array exists in allocated memory owned by the widget. The application must not modify or free this memory. The array will be freed as soon as the callback returns, so if the application needs to retain it, it must make a private copy.

XmNactivateCallback

The list of callback procedures to be called when the user double-clicks on an item in the legend. The call_data argument is of type XiActivateCallbackStruct* and points to a structure that contains a pointer to the selected or deselected plot, as well as an array of all currently selected plots. This callback structure is the same as the XiSelectCallbackStruct used by the XmNselectCallback, but the fields are used slightly differently: when a plot is "activated", the selected field will always be False, and the item field will always be set to the plot that was double clicked on. The selected_plots and num_selected_plots fields are exactly as for the XmNselectCallback; see the description above for cautions about using these fields.

In addition to these two callback lists, the Plotter has one additional resource that is applicable to selection in the Legend. Clicking in the legend will select any single item and deselect all others. By default, however, holding down the Shift key while clicking will select an item (or deselect a selected item) without changing the selection state of any other items. You can change this default with the XmNallowMultipleSelect resource:

XmNallowMultipleSelect

A Boolean that specifies whether the Plotter will allow multiple plots to be selected in the legend with shift-clicks. The default is True.

One last note about selecting items in the legend. To deselect all items in the legend, simply click on the legend title.

16.1.1 Example: Highlighting a Selected Plot

Example 16-1 shows a callback procedure you could use with the XmNselectCallback to highlight a selected plot using a special color. It assumes that all the plots in the Plotter are LinePlots, and that multiple-selection with shift-click is not allowed.

Example 16-1. Highlighting a selected plot

```
void HighlightLinePlot(Widget plotter,XtPointer tag,XtPointercall_data)
{
    XiSelectCallbackStruct *data = (XiSelectCallbackStruct) call_data;
    static Widget last_selected_plot = NULL;

    /* batch redisplays together */
    XiPlotterDisableRedisplay(plotter)

    /* First, unhighlight the last selected line plot */
    if (last_selected_plot) {
        XtVaSetValues(data->item,
                      XtVaTypedArg, XmNlineColor, XmRString, "black", 6,
                      NULL);
        last_selected_plot = NULL;
    }

    /* and now highlight the selected line plot, if there is one */
    if (data->selected) {
        XtVaSetValues(data->item,
                      XtVaTypedArg, XmNlineColor, XmRString, "red", 4,
                      NULL);
        last_selected_plot = data->item;
    }

    /* do the batch of redisplays */
    XiPlotterEnableRedisplay(plotter)
}
                .
                .
                .
XtAddCallback(plotter, XmNselectCallback, HighlightLinePlot, NULL);
```

16.2 Selecting Points

The user can select a point in the plotting area by clicking or double-clicking on it. The Plotter doesn't take any action of its own in response to these clicks, nor does it highlight the selected point in any way, but it does notify the application of the selection through the XmNclickCallback and XmNdoubleClickCallback callback lists. By default, these callbacks will be invoked when the user clicks or double-clicks either with no keys held down, or with the Shift key held down. While the user has the mouse button down during the click or double-click, the XmNclickCursor will briefly be displayed.

Both the XmNclickCallback and XmNdoubleClickCallback callbacks are invoked with a pointer to the following structure as their *call_data* argument:

```
/* Motif-style reason codes for these callbacks */
#define XmCR_CLICK          1 /* button clicked in plotting area */
#define XmCR_DOUBLE_CLICK    2 /* double click in plotting area */

typedef struct {
    int reason;        /* the type of the callback */
    XEvent *event;     /* the event that triggered it */
    Position px, py;   /* pixel coordinates of the mouse */
    double x, y;       /* plot coordinates of the mouse */
} XiClickCallbackStruct, XiDoubleClickCallbackStruct;
```

The callback lists that use this data structure are the following:

XmNclickCallback

The list of callback procedures to be called when the user clicks in the plotting area. The call_data argument is of type XiClickCallbackStruct* and points to a structure that contains the pixel coordinates (px and py) and axis coordinates (x and y) of the point over which the click occurred. Some applications may want to distinguish between a click and a shift-click; you can do this by examining the state field of the XButtonEvent * structure passed in the event field of the callback structure. If (state & ShiftMask) is non-zero, then the Shift key was held down for this click. See an Xlib reference manual for more information about X Event structures.

XmNdoubleClickCallback

The list of callback procedures to be called when the user double-clicks in the plotting area. The call_data argument is of type XiDoubleClickCallbackStruct* and points to a structure that contains the pixel and plot coordinates of the point over which the double-click occurred. The coordinates reported are actually those of the point under the pointer when the first click occurred--since the mouse is likely to move slightly during a double-click, the coordinates of the first click are more likely to be what the user was aiming at. You can check whether the Shift key was held down in the same way as described for the XmNclickCallback.

In addition to these two callback lists, the Plotter has one more resource that is applicable to this form of interaction:

XmNdragThreshold

Since it is very difficult for a user to hold the mouse perfectly still while clicking, this resource specifies a threshold value that allows the plotter to distinguish between a click and a drag of some sort. If the mouse moves further than the number of pixels specified by this resource between the time that the mouse button is pressed and the time that it is released, then the click is really a drag of some sort, and neither the click callback nor the double-click callback will be invoked. The default value is 4 pixels.

16.2.1 Example: Hit Detection

Hit detection is the process of determining what is under the mouse pointer when the user clicks (i.e. what the user "hit" with the mouse.) Example 16-2 uses the XmNdoubleClickCallback and the function XiBarPlotGetBar() (introduced in Chapter 13, *Annotating Plots*) to determine whether the user has double-clicked on a bar in a bar chart. The callback procedure shown here doesn't do anything once it determines which bar has been "hit", but an application that allows the user to edit plot data might then ask the user to click again in order to set a new position for the bar.

Example 16-2. Hit detection

```
void HitDetectBarPlot(Widget plotter, XtPointer tag, XtPointer
call_data)
{
    XiDoubleClickCallbackStruct *data;
    Widget barplot;
    int num_bars, i;
    double x0, y0, x1, y1, tmp;

    barplot = (Widget) tag; /* the bar plot to check for hits on */
    data = (XiDoubleClickCallbackStruct *)call_data;

    /* figure out how many bars in the bar plot */
    XtVaGetValues(barplot, XmNnumValues, &num_bars, NULL);

    /* Now check each bar */
    for(i=0; i < num_bars; i++) {
        /* get the bounds of the bar */
        XiBarPlotGetBar(barplot, i, &x0, &y0, &x1, &y1);

        /* in some cases y0 may be > y1. Check for this and swap */
        if (y0 > y1) {tmp = y0; y0 = y1; y1 = tmp;}

        /* Now see if the click was inside the bar */
        if ((data->x >= x0) && (data->x <= x1) &&
            (data->y >= y0) && (data->y <= y1)) break;
    }

    /* if i == num_bars, then no bar was clicked on,so just return */
    if (i == num_bars) return;

    /* Otherwise, we found a hit, so do whatever we do */
    Hit(barplot, i);
}
                 .
                 .
                 .
XtAddCallback(plotter, XmNdoubleClickCallback,
              HitDetectBarPlot, (XtPointer)barplot);
```

16.3 Crosshairs and Mouse Motion

When the user clicks mouse button 1 in the plotting area without the Shift or Meta keys pressed, the Plotter will draw "crosshairs": vertical and horizontal lines between the mouse pointer position and each edge of the plotting area. These crossed lines make it easy for the user to precisely read the axis coordinates of the point the mouse is over. While the user holds the mouse button down and crosshairs are being displayed, the Plotter also invokes the XmNmotionCallback each time the mouse moves. The application might use this callback notification to update a numeric display of the pointer position, for example. When the user releases the mouse button, the crosshairs are erased, and the XmNleaveCallback is invoked, which might serve to notify the application that it should erase its numeric display of the pointer position.

It is also possible to configure the Plotter to handle crosshairs differently. By adding some translations to the widget and setting the XmNpermanentCrosshair resource, you can have the Plotter widget display crosshairs and call the XmNmotionCallback whenever the pointer is inside the plotting area. Also, by setting XmNdisplayCrosshair to False, you can have the Plotter invoke the XmNmotionCallback and XmNleaveCallback to notify your application of mouse motion without ever actually drawing the crosshair.

The XmNmotionCallback and XmNleaveCallback are invoked, respectively, with pointers to the following two structures as their *call_data* arguments:

```
/* Motif-style callback reason codes */
#define XmCR_MOTION 9 /* the crosshairs moved */
#define XmCR_LEAVE 10 /* pointer has left plotting area */

typedef struct {
    int reason;        /* the type of the callback */
    XEvent *event;     /* the event that triggered it */
    Position px, py;   /* pixel coordinates of the mouse */
    double x, y;       /* plot coordinates of the mouse */
} XiMotionCallbackStruct;

typedef struct {
    int reason;        /* the type of the callback */
    XEvent *event;     /* the event that triggered it */
} XiLeaveCallbackStruct;
```

The callbacks themselves are the following:

XmNmotionCallback

The list of callback procedures to be invoked each time the mouse moves while crosshairs are being displayed (or even when crosshairs are not being displayed, if XmNdisplayCrosshair is False). The XiMotionCallbackStruct Callback structure is the same one passed to the XmNclickCallback: the px and py fields specify the pixel position of the mouse pointer, and the x and y fields specify the same position in axis coordinates. This XmNmotionCallback list may be invoked many times in rapid succession, so the procedures registered on it should not do any time-intensive computations, or the Plotter widget will not be able to keep its crosshairs displayed in sync with the mouse pointer.

XmNleaveCallback

The list of callback procedures to be invoked whenever crosshairs are erased--when the user releases the mouse button or, for permanent crosshairs, when the mouse pointer leaves the plotting area. Note that the XiLeaveCallbackStruct has no fields other than the standard reason and event fields.

A number of other resources control the reporting of motion events, and how, and whether crosshairs are drawn:

XmNallowCrosshair

This resource is a Boolean that specifies whether crosshairs should ever be displayed when the user clicks and drags the mouse in the plotting area. The default is True. If you set this resource to False, then no crosshairs will be displayed, and the XmNmotionCallback will never be invoked in response to a dragging the mouse. Note however, that even if XmNallowCrosshair is False, crosshairs may still be displayed in XmNpermanentCrosshair (documented below) is True.

XmNdisplayCrosshair

This resource specifies whether crosshairs should actually be drawn and updated each time the XmNmotionCallback is invoked. The default is True. Some applications will want to receive notification of user motion events (i.e. they will have either XmNallowCrosshair or XmNpermanentCrosshair set to True) but they will not actually want the crosshairs to appear on the screen. These applications can set this XmNdisplayCrosshair to False to have the Plotter invoke the XmNmotionCallback and XmNleaveCallback without ever drawing the crosshairs.

XmNcrosshairLineWidth

The width, in pixels, of the lines used to draw the crosshairs. The default of 0 specifies a thin 1-pixel line width which is ideal for accurate reading of the current position on the axes. In some cases, however, a 3-pixel width will be more legible. (A 3 pixel line will be centered about the correct point; a 2 pixel line may make accurate readings more difficult.)

XmNcrosshairColor

The color of the crosshairs to be displayed. On color displays, you should set this resource to a value that is distinct from the axis grid lines. Note that the crosshairs are drawn with XOR, which means that this color will only appear where the crosshairs overlap the plotting area background color--when the crosshairs cross a grid line or plot, other, seemingly random, colors will appear (unless you have allocated your colors very carefully.)

XmNcrosshairCursor

The pointer cursor to be displayed while the crosshairs are displayed. The default value of this resource is the cursor named "crosshair".

XmNpermanentCrosshair

Setting this resource to True tells the Plotter widget that you want crosshairs displayed and the XmNmotionCallback invoked whenever the pointer is within the plotting region, even when no mouse buttons are down--i.e. you want crosshairs displayed "permanently" When in "permanent crosshair" mode, the crosshairs will be erased and the XmNleaveCallback will be invoked only when the mouse leaves the plotting area. Setting XmNpermanentCrosshair is not actually sufficient to turn on permanent crosshairs, however; you must also add the following translations to the Plotter widget (use XtParseTranslationTable() and XtOverrideTranslations(), or use the #override directive and specify these translations in a resource file):

```
<Motion>: motion()
<Leave>: leave()
```

Note that these translations require the X server to sent an X event to your application any time the mouse moves anywhere within the Plotter widget. This can increase X Protocol traffic between your application and server significantly, and can result in poor performance with X terminals and other situations in which the server and application are not running on the same host. See Section 16.7, *Plotter Translations and Actions* for more information on Plotter translations and actions.

A "permanent crosshair" will be displayed even if XmNallowCrosshair is False--that resource only applied to crosshairs displayed in response to mouse drags. Note, though, that if XmNdisplayCrosshair is False a permanent crosshair will never be drawn, though the XmNmotionCallback and XmNleaveCallback will be invoked as if it were.

16.3.1 Example: Displaying Mouse Coordinates

Example 16-3 shows a callback procedure you could use with the XmNmotionCallback and XmNleaveCallback to display the coordinates of the mouse in a Motif XmLabel widget.

Example 16-3. Displaying Mouse Coordinates

```
void DisplayMouseCoords(Widget plotter, XtPointer tag,
XtPoinintercall_data)
{
    Widget labelwidget = (Widget) tag; /* the widget to display in */
    XiMotionCallbackStruct *data = (XiMotionCallbackStruct
*)call_data;
    int reason;
    char coords[50];
    XmString label;

    /* First go see if this is a motion callback of any kind */
    reason = ((XmAnyCallbackStruct *)call_data)->reason;
    if (reason == XmCR_MOTION) {
/*
 * If so, figure out the numbers to display, convert to
 * an XmString and set on the label widget.
 */
sprintf(coords, "X: %6.3f; Y: %6.3f", data->x, data->y);
label = XmStringCreateLocalized(coords);
XtVaSetValues(labelwidget, XmNlabelString, label, NULL);
XmStringFree(label);
    }
    else /* otherwise, just clear the label widget and we're done */
        XtVaSetValues(labelwidget, XmNlabelString, NULL, NULL);
}
                .
                .
                .
XtAddCallback(plotter, XmNmotionCallback,
              DisplayMouseCoords, (XtPointer) coord_label);
XtAddCallback(plotter, XmNleaveCallback,
              DisplayMouseCoords, (XtPointer) coord_label);
```

16.4 Selecting Regions

When the user holds down the **Ctrl** key and drags with mouse button 1 in the plotting area, the plotter displays special mouse cursors and "rubberbands" out a rectangle to indicate the bounds of the selected region. While the drag is in progress, the Plotter reports all mouse motion to the application by invoking the XmNdragMotionCallback. When the user releases the mouse button, the Plotter erases the rubberbanded rectangle, and reports the coordinates of the selected region through the XmNdragCallback. The user can cancel a drag in progress by clicking the right mouse button.

An application might use this style of interaction to allow the user to "zoom-in" on a plot--when the XmNdragCallback is invoked, the application would set new bounds on the X and Y axes to match the selected region. The same application might use the XmNdragMotionCallback to update a Label widget that displays the current bounds of the selected region numerically.

Besides these two callback lists, the Plotter has several other resources that control this kind of user interaction. These resources will be documented below. (Note that the callbacks and resources that are used for this type of user interaction all have the word "drag" in their names. These may be misleading names--there are several other types of user interaction that involve dragging with mouse.)

Both the XmNdragCallback and the XmNdragMotionCallback are passed a pointer to the following structure in their *call_data* argument:

```
/* Motif-style callback reason codes */
#define XmCR_DRAG            3 /* end of drag in plotting area */
#define XmCR_DRAG_MOTION     4 /* motion during drag in plotting
area */

typedef struct {
    int reason;        /* the type of the callback */
    XEvent *event;     /* the event that triggered it */
    Position px1, py1; /* pixel coords of upper left */
    Position px2, py2; /* pixel coords of lower right */
    double x1, y1;     /* plot coords of upper left */
    double x2, y2;     /* plot coords of lower right */
} XiDragCallbackStruct, XiDragMotionCallbackStruct;
```

The callbacks are the following:

XmNdragCallback

The list of callback procedures to be called when the user drags and releases the mouse over the plotting area. The call_data argument is of type XiDragCallback-Struct* and points to a structure that contains the coordinates of the upper left and lower right corners of the rectangle that was dragged out. The px1, py1 (upper-left), px2, and py2 (lower-right) fields give these coordinates as pixel values, and the x1, y1, x2, and y2 fields give these coordinates as axis values.

XmNdragMotionCallback

The list of callback procedures to be called for each mouse motion event that occurs while the mouse is being dragged over the plotting area to select a region. The call_data argument is of type XiDragCallbackStruct* and points to a structure that contains the pixel and plot coordinates of the upper left and lower right corners of the rectangle that was dragged out. This callback might be used to display the coordinates of the rectangle in the application's mode line, for example, but should not perform extensive computation, or the response time of the plotter rubberbanding will be degraded.

The other resources that are applicable to this type of user interaction are the following:

XmNallowDrag

A Boolean that specifies whether the Plotter will allow drags in the plotting area. The default is True. If your application does not register a callback to respond to drags, you should probably set this resource to False. This will stop the Plotter from rubberbanding when the user drags, and prevent confusion that could arise from a rubberbanded box that has no effect.

XmNdragLineWidth

This is the width, in pixels, of the lines drawn by the Plotter for the drag rubberband box. The default is 0, which is a special case for efficiently drawn one-pixel wide lines. Since the rubberband box is drawn using XOR mode, it can be difficult to see a one-pixel wide line when it is directly on top of a grid line, or overlaps a shaded bar plot or pie plot. Using lines that are two pixels wide can help to make the box more visible.

XmNdragColor

This is the color of the rubberband which is drawn while dragging. If it is not set when the widget is created, its value is inherited from the XmNforeground resource. The grid lines drawn by the axes default to this same color, and on color displays, it is useful to set these to different colors, so that the rubberband is visible even when it is directly over a grid line. Note that the "rubberband" box drawn while the drag is in progress is drawn with XOR mode, the color you request with the resource will only appear where the "rubberband" is directly over the plotting area background color (XmNplotAreaColor)--where the rubberband intersects grid lines and plots, other, somewhat random, colors will appear.

XmNtopLeftCursor, XmNtopRightCursor, XmNbottomLeftCursor, XmNbottomRightCursor

These resources specify the four cursors to be used when the user drags out a rectangle. When the user is dragging out the upper-right corner of the rectangle, the XmNtopRightCursor will be displayed, for example. The default values for these cursor resources are right angles that match the appropriate corner of the rectangle. If you change one of these resources, you will probably want to change them all to provide a matching set.

16.4.1 Example: Zooming In

You can use the selected region reported by the XmNdragCallback in many ways. Perhaps the most natural one, however is to zoom in on the selected area. Example 16-4 shows a callback procedure that does just this. It also shows another callback procedure that uses the Plotter autoscaling feature to restore the Plotter display to its original size. (These procedures assume that your plots are displaying a fixed set of data, and that you are not computing new data at a higher resolution when the user zooms in.) Note that you could also write a function similar to the DisplayMouseCoords() shown in Example 16-3 for use with the XmNdragMotionCallback.

Example 16-4. Zooming In

```
void ZoomIn(Widget plotter, XtPointer tag, XtPointer call_data)
{
    XiDragCallbackStruct *data = (XiDragCallbackStruct *) call_data;
    Widget xaxis = XiPlotterXAxis(plotter);
    Widget yaxis = XiPlotterYAxis(plotter);

    /* Set the bounds on the axes, and batch the changes together */
    XiPlotterDisableRedisplay(plotter);
    XiAxisSetBounds(xaxis, data->x1, data->x2);
    XiAxisSetBounds(yaxis, data->y2, data->y1);
    XiPlotterEnableRedisplay(plotter);
}

/* A callback to register on a push button in a menu */
void ZoomRestore(Widget w, XtPointer tag, XtPointer call_data)
{
    Widget plotter = (Widget) tag;  /* the plotter to zoom out */
    Widget xaxis = XiPlotterXAxis(plotter);
    Widget yaxis = XiPlotterYAxis(plotter);

    /* Set autoscaling on the axes, and batch the changes together */
    XiPlotterDisableRedisplay(plotter);
    XiAxisAutoscale(xaxis);
    XiAxisAutoscale(yaxis);
    XiPlotterEnableRedisplay(plotter);
}
.
.
.
XtAddCallback(plotter, XmNdragCallback, ZoomIn, NULL);
XtAddCallback(restore, XmNactivateCallback, ZoomRestore,
(XtPointer)plotter);
```

16.5 Panning or Scrolling

When the user holds down the Meta (sometimes Alt) key and drags with mouse button 1 in the plotting area, the Plotter displays a special mouse cursor, and draws the outline of a large rectangle in the plotting area. This rectangle represents the plotting area itself, and moves as the user moves the mouse--this is an intuitive way to allow the user to pan (i.e. scroll) the visible area of the plot left, right, up, or down. While the interaction is in progress, the Plotter reports all mouse motion events through the XmNpanMotionCallback, and when the user releases the mouse button, the Plotter reports the final delta-X and delta-Y motion through the XmNpanCallback. The user can cancel the interaction in progress by clicking the right mouse button.

Note that the Plotter does not actually adjust the bounds of the axes in response to a to this user interaction, but the application can easily do this in response to the XmNpanCallback. The application might also use the XmNpanMotionCallback to update a numeric display of the adjusted coordinates, for example.

The two callbacks are passed a pointer to the following structure as their *call_data*:

```
/* Motif-style callback reason codes */
#define XmCR_PAN             5 /* end of pan in plotting area */
#define XmCR_PAN_MOTION     6 /* motion during pan in plotting area
*/

typedef struct {
    int reason;        /* the type of the callback */
    XEvent *event;     /* the event that triggered it */
    Position pdx, pdy; /* delta x and delta y in pixel coords */
    double dx, dy;     /* delta x and delta y in plot coords */
} XiPanCallbackStruct, XiPanMotionCallbackStruct;
```

The callbacks are the following:

XmNpanCallback

The list of callback procedures to be called when the user releases the mouse after panning in the plotting area. The call_data argument is of type XiPanCallbackStruct* and points to a structure that contains the delta-x and delta-y of the pan, in both pixel and axis coordinates.

XmNpanMotionCallback

The list of callback procedures to be called for each mouse motion event that occurs while the user pans over the plotting area. The call_data argument is of type XiPanMotionCallbackStruct* and points to a structure that contains the delta-x and delta-y of the pan, in both pixel and axis coordinates. This callback might be used to display the size of the pan in the application's mode line, for example, but should not perform extensive computation, or the response time of the plotter will be degraded and the panning rectangle will be updated jerkily.

Besides these two callbacks, several other resources control this type of user interaction:

XmNallowPan

A Boolean resource that specifies whether the Plotter will respond to panning in the plotting area. The default is True, but if your application does not register any callbacks to handle panning you should probably set this resource to False. This will stop the Plotter from displaying the panning rectangle, and will prevent the confusion that could result from an apparent pan that had no effect.

XmNpanLineWidth

The width, in pixels, of the lines to be used to draw the pan rectangle. The default is 0 which is a special value meaning efficient one-pixel wide lines. As with the XmNdragLineWidth resource, the pan rectangle will be more visible when overlapping grid lines if this resource is set to 2.

XmNpanColor

The color of the pan rectangle. If this value is not set when the widget is created, it is inherited from the XmNforeground resource of the Plotter, and as with the XmNdragColor resource, it is useful to set this to a distinct color on color displays.

XmNpanCursor

The cursor to display when the user pans the widget. The default is the cursor named "fleur" in the standard cursor font. (This cursor displays arrows pointing in each of the four cardinal directions.)

16.5.1 Example: Scrolling the Plotting Area

The panning described in this section is a specialized enough interaction that the only callback you are likely to want to register on the XmNpanCallback is one that scrolls the plotting area.Example 16-5 shows how you might implement such a callback.

Example 16-5. Scrolling the plotting area

```
void Scroll(Widget plotter, XtPointer tag, XtPointer call_data)
{
    XiPanCallbackStruct *data = (XiPanCallbackStruct *) call_data;
    Widget xaxis = XiPlotterXAxis(plotter);
    Widget yaxis = XiPlotterYAxis(plotter);
    double xmin, xmax, ymin, ymax;

    /* First, get the current bounds of the axes */
    XiAxisGetBounds(xaxis, &xmin, &xmax);
    XiAxisGetBounds(yaxis, &ymin, &ymax);

    /* Then set the new bounds on the axes, and batch the changes
       together */
    XiPlotterDisableRedisplay(plotter);
    XiAxisSetBounds(xaxis, xmin + data->dx, xmax + data->dx);
    XiAxisSetBounds(yaxis, ymin + data->dy, ymax + data->dy);
    XiPlotterEnableRedisplay(plotter);
}
.
.
.
XtAddCallback(plotter, XmNpanCallback, Scroll, NULL);
```

16.6 Selecting Title and Axes

When the user clicks or double-clicks with mouse button 1 (and optionally with the Shift key down) on the Plotter title or on any of the Axes, the Plotter notifies the application of this interaction by invoking the XmNeditCallback. The application might use this notification to pop up a dialog box that allows the user to edit the attributes of the selected axis, for example.

The XmNeditCallback is invoked with a pointer to the following structure as its *call_data* argument:

```
#define XmCR_EDIT          11 /* A Motif-style callback reason code
*/

#define XiPlotterEditTitle 1   /* Values for the where field below
*/
#define XiPlotterEditXAxis  2
#define XiPlotterEditYAxis 3
#define XiPlotterEditYAxis2 4

typedef struct {
    int reason;
    XEvent *event;
    int where;            /* one of the above constants */
    Boolean double_click; /* True if this was a double-click */
} XiEditCallbackStruct;
```

XmNeditCallback

The list of callback procedures to be invoked when the user clicks or double-clicks on the Plotter title or on any of its axes. The where field indicates what the user clicked on, and is one of the four constants shown above. The double_click field indicates whether the user clicked or double-clicked--if True, then the user double-clicked. You can determine whether the Shift key was held down during the click or double-click by casting the event field to an XButtonEvent * and examining the state field of that event structure to see if the ShiftMask flag is set.

16.7 Plotter Translations and Actions

In the X Toolkit, the user's mouse and keyboard events do not directly trigger the callback lists to be invoked. Instead, the X Toolkit looks up these user events in the "translation table" of the appropriate widget and maps the events to named "action procedures" defined by the widget. The Plotter widget uses this "translations and actions" scheme, and so by changing the default translation table it is possible to modify the mapping of user events to widget actions and effectively change the widget's key and mouse-button bindings. In order to do this, you need to understand the Plotter's default translation table and each of the action procedures it supports.

Figure 16-1 shows the default Plotter translations.

As translation tables go, these default Plotter translations are exceptionally confusing. This is because so many different kinds of interactions are supported through the same basic mouse button 1 click-and-drag interaction. You need to understand the individual action procedures in detail to make sense of these translations. As the actions are explained below, you may find it useful to refer back to Figure 16-1 to see how the actions are used.

arm()

This action is used with the activate() action, and together, the pair of actions is responsible for selection in the legend, and selection of the Plotter title and axes. The arm() action must be bound to a button-down event of some sort. The arm() action will ignore any events that take place inside the plotting area of the Plotter. This means that you can safely bind it to the same event as the begin() action--begin() ignores any button down events that occur *outside* of the plotting area.

```
/*
 * These default translations are the following:
 *   unmodified button 1 is for clicking, or displays crosshairs if
     dragged.
 *   shift button 1 is for clicking, in case the app wants to use
     shift-click
 *   ctrl-button 1 is for dragging a rectangle.
 *   meta-button 1 is for panning.
 *   button 1 over the legend selects or activates (double-click) an
     item
 *   shift-button1 over the legend selects without deselecting others.
 *
 * These can easily be bound in other ways. If using other buttons,
 * then don't forget to bind motion() and end() to button motion and
 * button up for those buttons too.
 *
 * These translations will not work with a permanent crosshair. For
 * that, we also need these two:
 *    <Motion>: motion()
 *    <Leave>:   leave()
 *
 * The begin() and arm() actions, and the end() and activate() actions
 * are designed not to interfere with each other; only one will have
 * any effect for any given event.
 */
static char defaultTranslations[] = "\
    Shift<Btn1Down>:    begin(click) arm()\n\
    Ctrl<Btn1Down>:     begin(drag)\n\
    Meta<Btn1Down>:     begin(pan)\n\
    <Btn1Down>:         begin(click,crosshair) arm()\n\
    <Btn1Motion>:       motion()\n\
    Shift <Btn1Up>:     end() activate(multiple)\n\
    <Btn1Up>:           end() activate()\n\
    <Btn3Down>:         cancel()\n\
    <Key>Escape:        cancel()\n\
    <Key>osfCancel:     cancel()";
```

Figure 16-1. The default translation table of the Plotter widget

activate()

This action must be invoked after the arm() action, and is responsible for calling the XmNselectCallback, XmNactivateCallback, and XmNeditCallback. activate() must be bound to a button-up event that corresponds to a button-down event to which the arm() action is bound.

The activate() action takes an optional argument--if you pass the argument "multiple", then clicking in the legend will toggle the selection of the item clicked on and will not deselect other items. Without the "multiple" argument, clicking in the legend selects the item clicked on an deselects all others. The default translations bind arm() to <Btn1Up>, and arm(multiple) to Shift<Btn1Up> to allow both single and multiple selection. By modifying these translations you can allow only single or only multiple selection.

Note that there are no special arm() or activate() bindings in the translation table for double-clicks. The activate() action keeps track of the time it was last invoked and checks the elapsed time to determine whether a given event is a click or a double click. It uses the standard "system" multi-click interval, which the user can set with the Xt application resource multiClickTime, and which you can set with XtSetMultiClickTime() before you create the Plotter widget.

The activate() action ignores any button-up events if the corresponding button-down event occurred within the Plotter's plotting area. This means you can safely use it with the end() action which ignores all events that occur outside of the plotting area.

begin()

The begin() action is used with the motion() and end() actions, and together, this trio of actions is responsible for almost all user interaction that takes place within the plotting area. begin() must be bound to a button-down event of some sort. When invoked for an event that occurs outside of the plotting area, it ignores the event, which means that you can safely bind it to the same events as the arm() action.

The begin(), motion(), and end() actions handle four distinct types of user input: clicks and double-clicks, crosshairs displayed while the user drags the mouse (but not "permanent crosshairs"), region selection with a rubberbanded box ("dragging"), and panning. The begin() action doesn't invoke any callbacks, but does get the plotter ready to perform these different kinds of interaction when subsequent mouse motion and button-up events arrive. In order to do this, begin() takes an argument that specifies what kind of input is "beginning". The argument may be one of the strings "click", "crosshair", "drag" or "pan", which correspond to the four types of user input described above. (Note that most of these input types involve dragging the mouse. The "drag" argument really means "select region".)

The "click" argument is a special one: when the user presses a mouse button, there is no way of knowing whether she intends to release the button right away, resulting in a click, or whether she intends to drag the mouse before releasing the button. For this reason, if you specify the "click" argument to begin(), you may also specify an optional second argument of "crosshair", "drag", or "pan" which specifies what to do if the user ends up dragging the mouse instead of clicking. The default translation table, for example, binds begin(click,crosshair) to <Btn1Down>, which means that if the user presses and releases button 1, the XmNclickCallback will be invoked, but, instead, if the user presses the button, and then moves the mouse further than XmNdragThreshold, crosshairs will appear instead, and the Plotter will begin to call XmNmotionCallback for each mouse motion.

motion()

The motion() action is used after the begin() action and before the end() action. It must be bound to a button motion event of some kind. The behavior of this action will depend on the argument or arguments passed to the previous invocation of the begin() action. motion() is responsible for updating the crosshairs, region select rectangle or pan rectangle that the Plotter is displaying, and for calling the XmNmotionCallback, the XmNdragMotionCallback, or the XmNpanMotionCallback().

Usually the motion() action is bound to button motion events, so it is only invoked when the user moves the mouse while holding down a mouse button. But when you want permanent crosshairs displayed, you must also bind this action to the <Motion> event, so that the crosshairs are updated whenever the mouse moves.

end()

This action is used after the begin() and motion() actions have been invoked. It must be bound to a button-up event of some kind, and its behavior depends on the argument or arguments passed to the previous invocation of the begin() action. end() is responsible for erasing the crosshairs, region select rectangle or pan rectangle drawn by motion(), and for calling the appropriate callback: one of XmNclickCallback, XmNdoubleClickCallback, XmNleaveCallback, XmNdragCallback, or XmNpanCallback. Note that no special bindings are required to handle double-clicks. The end() action distinguishes clicks from double-clicks in the same way that the activate() action does.

The end() action ignores any button-up events for which the corresponding button-down event occurred outside of the plotting area, which means that you can safely bind it to the same events as the activate() action, which ignores all events inside of the plotting area.

leave()

The leave() action is only necessary when you are displaying permanent crosshairs. It is responsible for erasing the crosshairs and invoking the XmNleaveCallback. This action is not bound in the default translation table, but when using permanent crosshairs, you should bind it to the <Leave> event.

cancel()

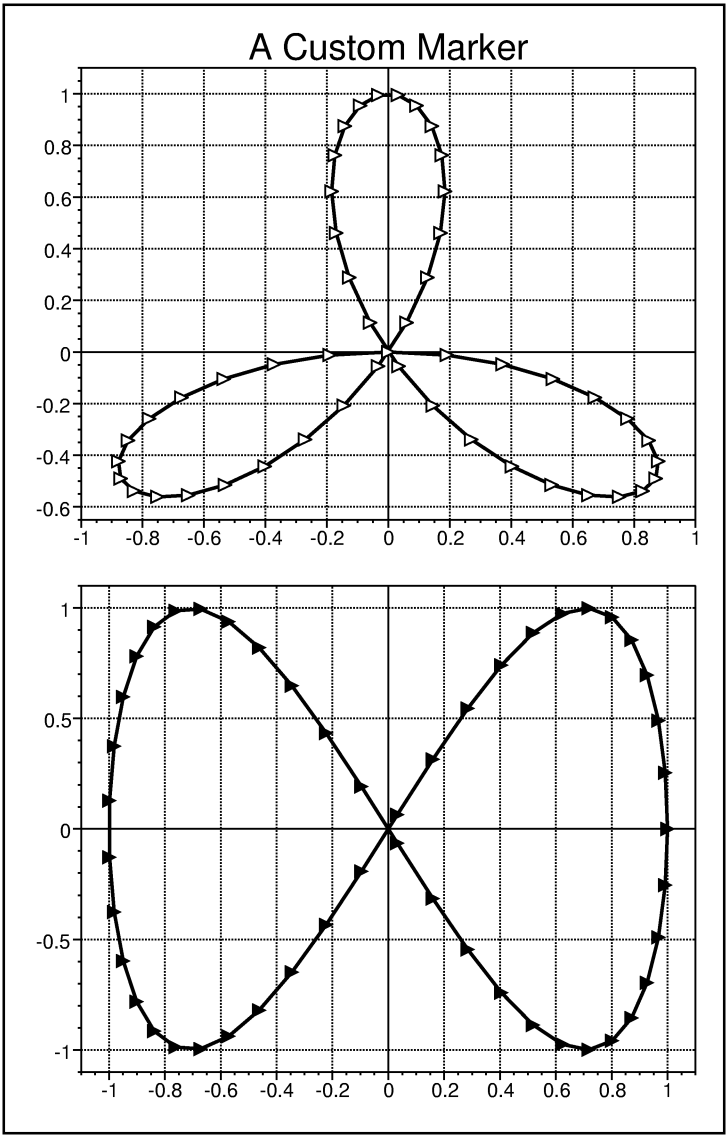
This action gives the user a way to cancel an interaction in progress without having any of the callbacks called. Suppose the user has pressed a mouse button and started dragging to select a region of the plotting area, for example, and then they decide that they don't want to zoom in on that region (assuming the application does a zoom in in response to the XmNdragCallback). By invoking the cancel() action they can terminate the action in progress without having any callbacks called. cancel() works to terminate any kind of user input in progress. In the default translation table, it is bound to mouse button 3, and also to the Escape key. Note, however, that Motif applications grab Escape for their own purposes, so if you are using Motif, the Escape binding will not work.

16.7.1 Example Translation Tables

The default Plotter translation table is only one possible scheme for binding events to actions. Example 16-6 shows some other translation tables that you might specify in a resource file.

Example 16-6. Example Translation Tables

```
!!
!! Use Button 1 for clicking, Button 2 for crosshairs and
!! Button 3 for selecting regions. Panning is not allowed.
!! shift-click in the legend does multiple select, and Ctrl-G cancels.
!! (Note: for X11R4, use .translations instead of .baseTranslations)
!!
*XiPlotter.baseTranslations: \
        <Btn1Down>:     begin(click) arm()\n\
        <Btn1Motion>:   bac\n\
        Shift <Btn1Up>: end() activate(multiple)\n\
        <Btn1Up>:       end() activate()\n\
        <Btn2Down>:     begin(crosshair)\n\
        <Btn2Motion>:   motion()\n\
        <Btn2Up>:       end()\n\
        <Btn3Down>:     begin(drag)\n\
        <Btn3Motion>:   motion()\n\
        <Btn3Up>:       end()\n\
        Ctrl<Key>G:     cancel()

!!
!! Display crosshairs all the time.
!! Button 1 does clicks & region selection as well as legend selection
!! Shift-Button1 does panning and multiple legend selection
!!
*XiPlotter.permanentCrosshair: True
*XiPlotter.translations: \
        <Motion>:               motion()\n\
        <Leave>:                leave()\n\
        Shift<Btn1Down>:        begin(pan) arm()\n\
        <Btn1Down>:             begin(click,drag) arm()\n\
        <Btn1Motion>:           motion()\n\
        Shift <Btn1Up>:         end() activate(multiple)\n\
        <Btn1Up>:               end() activate()\n\
        <Btn3Down>:             cancel()
```


A Custom Marker
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1
1
0.5
0
-0.5
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

17

Defining Custom Markers

As explained in Chapter 3, *Plotter Data Types and Resources*, the XiMarker is an abstraction that is capable of displaying a small glyph or "marker" both in X and in PostScript. The XiPlotter library provides 12 standard markers, each in four different sizes. These pre-defined markers are suitable for most purposes, but there may be times when you will want to display a plot using some custom glyph to mark its points. This chapter explains how to create your own XiMarker for use in X, how to add PostScript support to your markers, and how to register a name for your custom markers, so they can be used through the XiMarker resource converter.

17.1 Creating a Custom Marker

You can create your own custom markers with XiMarkerCreate(), shown in Figure 17-1.

The first two arguments to this function are the width and height of the marker, in pixels. The third argument, mark_bits, is an array of characters that define the marker in XBM format. This is generally an array of characters defined in a XBM file produced with the bitmap program that is part of the standard X distribution.

The fourth argument, mask_bits, is the "mask" for the marker. It is also an array of characters that define a bitmap in XBM format. There is an important difference between the marker bitmap and the mask bitmap, however. The mark_bits define which bits of the marker should be drawn with the foreground color, and which should be drawn in the background color, but the mask_bits define which bits should be drawn at all. It is this "mask" which defines the overall shape of the marker. If a bit is set in the mask, then the corresponding pixel on the screen will be drawn in either the foreground or the background color, depending on the bits in the marker bitmap. If a bit is not set in the mask, then the corresponding pixel on the screen will not be modified; in effect, those unset bits are not part of the marker at all. A mask like this is necessary for any marker that is not rectangular; the same technique is used to define mouse cursors.Figure 17-2 shows a marker and a mask bitmap that might be used to define a custom marker.

```
XiMarker XiMarkerCreate(int width, int height,
                          char *mark_bits, char *mask_bits,
                          String psmark, String psmask);
```

`width, height`

The size of the new marker in pixels.

`mark_bits, mask_bits`

XBM bitmap data for the marker and its mask.

`psmark, psmask`

PostScript language descriptions of how to draw the marker and its mask.

`Returns`

The newly created XiMarker.

Figure 17-1. XiMarkerCreate()

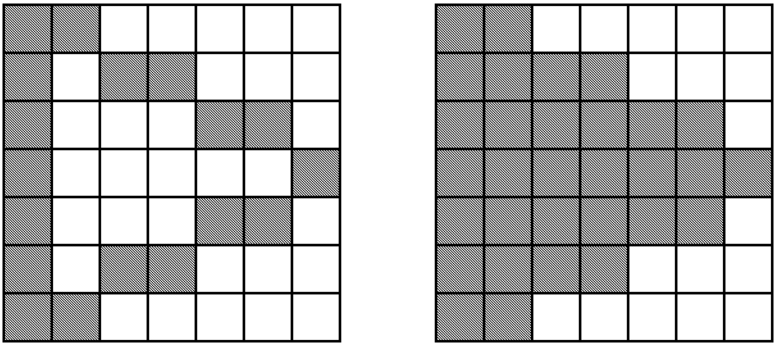


Figure 17-2. Marker and mask bitmaps for a custom XiMarker

Note that the marker bitmap is a hollow triangle, and the mask bitmap is a filled triangle. This means that the bits inside the triangle will be drawn with the background color, and the resulting marker will be opaque--when it is drawn over a line, that line will not show through the hollow center. If the mask bitmap were the same as the mark bitmap, then only the edges of the triangle would be part of the marker, and the marker would not be opaque.

Having defined mark and mask bitmaps, we can define our custom triangle marker as shown in Example 17-1.

Example 17-1. Creating a custom marker (X only)

```
/*
 * the following lines of code are the output of the standard
 * X 'bitmap' application. We could also just have included these
 * bitmaps files with #include
 */
#define triangle_width 9
#define triangle_height 9
static char triangle_bits[] = {
   0x03, 0x00, 0x0d, 0x00, 0x31, 0x00, 0xc1, 0x00, 0x01, 0x01, 0xc1,
   0x00, 0x31, 0x00, 0x0d, 0x00, 0x03, 0x00};

#define triangle_mask_width 9
#define triangle_mask_height 9
static char triangle_mask_bits[] = {
   0x03, 0x00, 0x0f, 0x00, 0x3f, 0x00, 0xff, 0x00, 0xff, 0x01, 0xff,
   0x00, 0x3f, 0x00, 0x0f, 0x00, 0x03, 0x00};

XiMarker triangle;

triangle = XiMarkerCreate(triangle_width, triangle_height,
                           triangle_bits, triangle_mask_bits,
                           NULL, NULL);
```

This custom marker will work on-screen, but will not print. Getting a marker to print in Plotter PostScript output requires the remaining two arguments to XiCreateMarker(), and is the subject of the next section.

17.2 PostScript Output of Custom Markers

To define a marker that can be printed, you must supply the two final arguments to XiMarkerCreate(). These arguments are strings that give the PostScript drawing commands for the marker itself, and for its mask. Since PostScript is a resolution-independent language, we cannot simply use a bitmap to draw the marker in PostScript; instead we must use PostScript drawing commands to actually draw it. Before you read the rest of this section, you should have at least an elementary understanding of the PostScript language. *The PostScript Language Reference Manual*, *Second Edition*, published by Addison-Wesley, is the definitive documentation on the language.

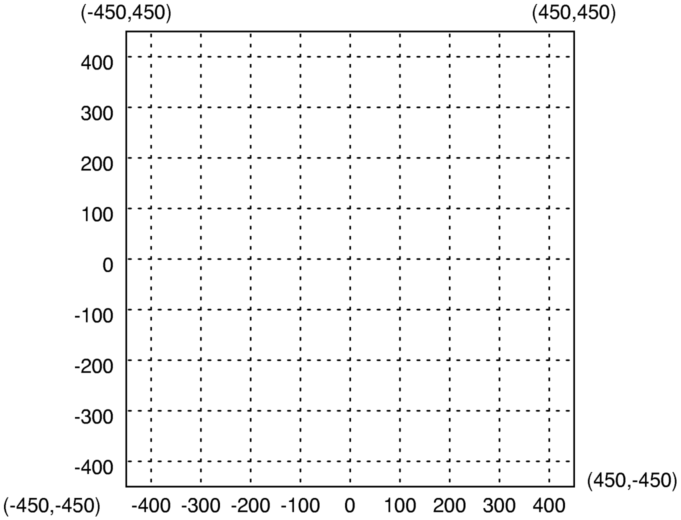


Figure 17-3. The bounding box and coordinate system for PostScript markers

The Plotter library defines markers as glyphs in a special marker font. The PostScript strings you provide must meet some special requirements to work with this font. The most noticeable of these requirements is the coordinate system. You must draw the glyph using a coordinate system with units 100 times as small as the corresponding pixels in the X bitmaps. So, to continue with our triangle marker example, since we specified a triangle that was 9 pixels wide by 9 pixels high, our PostScript triangle will be 900 units wide by 900 units high. (This doesn't mean we're defining a huge triangle, of course; just giving ourselves lots of resolution to play with. The Plotter library prints markers at approximately the same size that they appear on the screen.) Markers are always centered over the point they mark, so the bounding box for our PostScript version of the triangle is from -450 to 450 in the X dimension, and the same in the Y dimension. Also, note that in PostScript units increase upwards in the Y dimension; this is the reverse of the X coordinate system in which Y pixel coordinates increase as you move downwards. Figure 17-3 shows the coordinate system and the bounding box that the marker glyph and mask must be drawn within.

We must supply XiMarkerCreate() with a PostScript string that will draw our marker. Example 17-2 shows the PostScript code we will use, and the left hand side of Figure 17-4 shows the output of these commands. Remember that we are using the coordinate system pictured in Figure 17-3.

Example 17-2. PostScript code to draw the marker

```
    -400 375 moveto    % move to the upper left corner
    -400 -375 lineto   % draw a line to the lower left corner
    350 0 lineto        % draw a line to the vertex on the right
    closepath           % draw the remaining line
    100 setlinewidth   % specify a width for our lines
    stroke              % and actually draw the lines
```

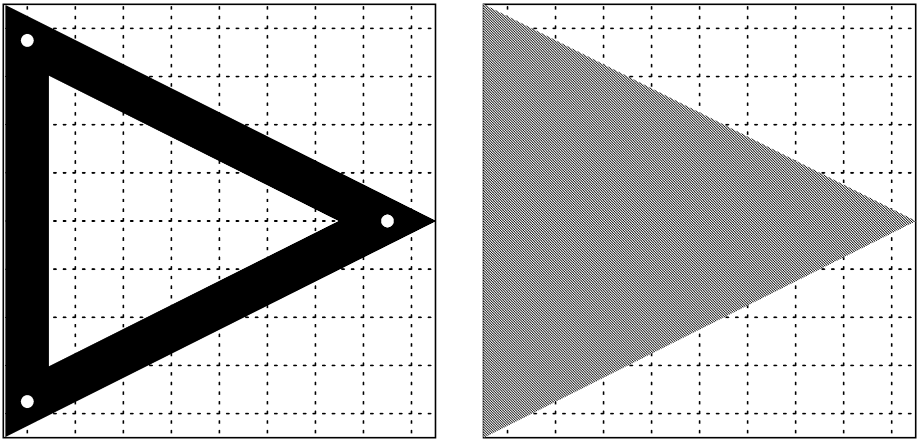


Figure 17-4. PostScript marker and mask

It might seem unusual to specify a linewidth of 100, but remember that we're working with an unusually detailed coordinate system. PostScript coordinate transforms affect all aspects of drawing, including line widths. With a line width of 100 units, our lines will extend for 50 units on either side of the coordinates we specify. For this reason, we've moved the vertices of the triangle in from the edges of the bounding box. We've moved them in more than 50 units in some cases, because when two line overlap in PostScript, they are joined with a "miter" that can be wider than the line width. (Careful trigonometric analysis and detailed graph paper can help when trying to figure out coordinates for PostScript glyphs like this. Trial and error works well too.)

Note that we do not use the gsave or grestore operators while drawing this glyph, even though we do things like changing the line width. The gsave and grestore are necessary, but are automatically handled by the font setup mechanism.

We'll make this PostScript string more compact by removing the comments, and by abbreviating the names of the PostScript operators. Figure 17-5 shows a standard set of abbreviations that is included in every PostScript file generated by the Plotter. You can use these abbreviations when defining markers. In particular, notice the definition of the WIL operator. It expects a single value on the stack, and sets it as the line width. It also sets all the other line attributes to their default values. These line attributes may be in an arbitrary state when your PostScript marker code is executed, so you must always set them to some known state. Usually, you can just use IL instead of setlinewidth as we do here.

```
%%BeginResource: procset XiPSAbbrevs 1.2 0
%%Copyright: (c) 1995 Integrated Computer Solutions, Inc.
/M /moveto load def
/L /lineto load def
/RM /rmoveto load def
/RL /rlineto load def
/ST /stroke load def
/CP /closepath load def
/FL /fill load def
/GS /gsave load def
/GR /grestore load def
/SG /setgray load def
/IT {transform round exch round exch itransform} bind def
/m {IT M} bind def
/l {IT L} bind def
/am {IT RM} bind def
/re {IT RL} bind def
/S {M show} bind def
/IL {setlinewidth [] 0 setdash 0 setlinecap 0 setlinejoin} bind def
/SW /setlinewidth load def
/SLC /setlinecap load def
/RF /rectfill load def   % or an emulation of rectfill for Level 1 PS
/RS /rectstroke load def % or an emulation of rectstroke for Level 1
PS
/RC /rectclip load def   % or an emulation of rectclip for Level 1 PS
%%EndResource
```

Figure 17-5. Standard PostScript abbreviations used by the Plotter

Using these abbreviations, and removing the comments, our PostScript marker definition becomes the string:

```
String psmark = "-400 375 M -400 -375 L 350 0 L CP 100 IL ST";
```

This is the PostScript code used to define the custom marker pictured in the graphs of the opening pages of this chapter.

XiMarkerCreate() also takes a psmask argument. This is another PostScript string, and serves a similar purpose to the bitmap mask that we defined for the screen version of the marker. In the X version of the marker, we used the mask to prevent pixels outside of the marker from being drawn. In PostScript, the problem is the reverse: we need a way to force pixels that are part of the marker to be drawn. In the PostScript code we defined above, only the pixels that are part of the three lines will be drawn in black. We want our marker to be opaque, so we need a way to "erase" the pixels on the inside of the triangle. We do this by defining a solid triangle as the mask--the Plotter library will draw that solid triangle in white and then draw our triangle outline in black; this erases anything that had previously been drawn under the marker.

We define our solid triangle with a fill command rather than a stroke command. Note that since we aren't drawing lines, we don't have to take line width into account, and can go all the way to the edges of the bounding box:

```
String psmask = "-450 450 M -450 -450 L 450 0 L FL";
```

The right-hand side of Figure 17-4 shows the results of these PostScript commands. This string just fills the triangle, and does not use setgray to set the fill color to white; this will be handled internally when the Plotter prints the marker. In fact, since both the marker and its mask are stored as glyphs in a PostScript font, they are not allowed to use the setgray operator, as this would break PostScript font caching. (Read about user defined fonts in the PostScript reference manual for more information on operators that font glyphs are not allowed to use.)

Example 17-3. Defining and using custom markers for X and PostScript

```
#include "markers/triangle"
#include "markers/triangle_mask"

static String psmark = "-400 375 M -400 -375 L 400 0 L CP 100 IL ST";
static String psmask = "-450 450 M -450 -450 L 450 0 L FL";

XiMarker triangle, ftriangle;

triangle = XiMarkerCreate(triangle_width, triangle_height,
                           triangle_bits, triangle_mask_bits,
                           psmark, psmask);

ftriangle = XiMarkerCreate(triangle_width, triangle_height,
                            triangle_mask_bits, triangle_mask_bits,
                            psmask, NULL);

XtVaSetValues(lineplot1, XmNmarker, triangle, NULL);
XtVaSetValues(lineplot2, XmNmarker, ftriangle, NULL);
```

With these PostScript strings defined, we can create custom markers that will display on the screen and print in PostScript.Example 17-4 shows code that does this. This example defines the hollow triangle marker we've been using as an example, and also reuses the X and PostScript masks to define a solid marker with the same shape. For this second marker, notice that the psmask string is optional--since the marker is already solid, there is nothing underneath it that needs to be erased first.

17.3 Naming and Looking Up Markers

When you have created a custom marker with XiMarkerCreate(), you can simply use that marker directly in your C code. You can also register the marker with a name, so that the String-to-Marker converter will recognize it and it can be used from resource files. You can do this with XiMarkerRegister() which takes a string and an XiMarker.

```
void XiMarkerRegister(String name, XiMarker marker);
```

`name` The name by which the marker is to be registered.

`marker` The XiMarker that is being registered.

```
XiMarker XiMarkerLookup(String name);
```

`name` The name of the marker to be looked up.

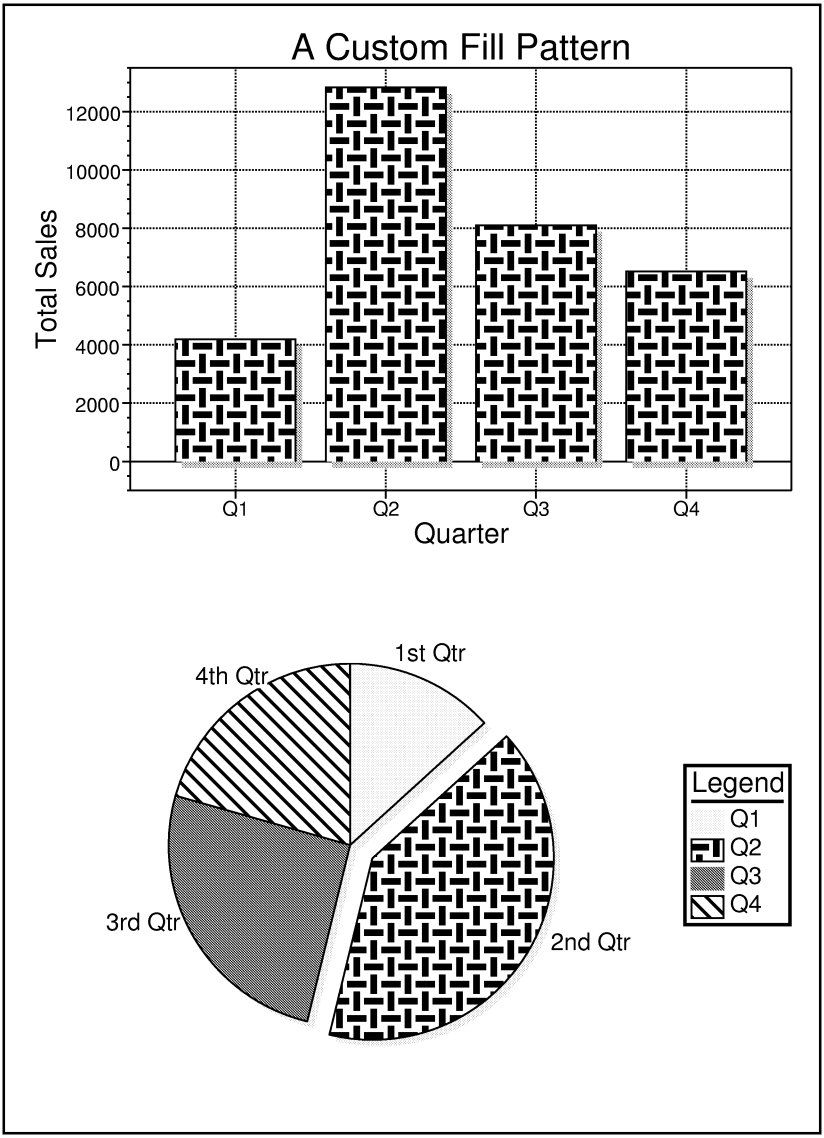
`Returns` A predefined or registered marker with the specified name, or NULL if no such marker is found.

Figure 17-6. XiMarkerRegister() and XiMarkerLookup()

We could register the markers we created in the previous example like this:

```
XiMarkerRegister("TriangleRight", triangle);
XiMarkerRegister("FilledTriangleRight", ftriangle);
```

The function XiMarkerLookup() looks up pre-defined and custom registered markers by name. You can use it when you want to lookup a marker by name without using the resource converter. This function looks up markers in exactly the same way that the resource converter does, and so recognizes the same predefined names.


A Custom Fill Pattern
Total Sales
12000
10000
8000
6000
4000
2000
0
Q1
Q2
Q3
Q4
Quarter
1st Qtr
4th Qtr
3rd Qtr
2nd Qtr
Legend
Q1
Q2
Q3
Q4

18

Defining Custom Fill Patterns

The BarPlot and PiePlot widgets optionally use fill patterns to shade their bars and pie wedges. The Plotter library provides 25 predefined patterns. Chapter 3, *Plotter Data Types and Resources,* explains how to use fill patterns, and a table of predefined patterns appears in the quick reference section at the end of this manual.

It is also possible to define your own custom patterns that will work with both X and PostScript. This chapter explains how to do that.

18.1 Creating a Custom Fill Pattern

You create a custom fill pattern with the function XiFillPatternCreate(), declared in <*Xi/Plotter.h*> and shown in Figure 18-1.

```
XiFillPattern XiFillPatternCreate(char *bits,
                                    int width, int height,
                                    String psglyph, String psfill);
```

`bits` XBM data that defines one cell of the fill pattern in X.

`width, height`

The size of the bitmap, in pixels.

`psglyph`

A string of PostScript commands to draw one cell of the pattern.

`psfill` An alternative PostScript string that sets parameters for the fill operator.

`Returns`

The newly created XiFillPattern.

Figure 18-1. XiFillPatternCreate()

The first three arguments to this function are required, and define a bitmap to be used as the fill pattern for X. The final two arguments are optional, but one or the other of them must be specified if the fill pattern is ever to be printed. The function returns a value of type XiFillPattern, which you can use to set resources, exactly as you would use any of the predefined fill patterns.

The following two subsections explain how to use XiFillPatternCreate() to create fill patterns that work with X and with PostScript.

18.1.1 Defining a Fill Pattern for X

The first three arguments to XiFillPatternCreate() are simply an XBM bitmap specification. XBM is the standard X bitmap format, and you can create XBM files using the standard X *bitma*p application. An XBM file defines a static array of char that are the raw data of the bitmap, and also provides symbolic definitions for the width and height of the bitmap. In X, the XiFillPattern abstraction uses this rectangular bitmap to tile the area to be filled. The *bitmap* client is not good at showing you what your bitmap will look like when used repeatedly to fill an area, but you can use *xsetroot -bitmap* for this purpose.

Figure 18-2 shows a bitmap you might create when defining a custom fill pattern, and Example 18-1 shows the code you'd use to create an XiFillPattern from the bitmap.

Example 18-1. Creating an XiFillPattern (X only)

```
/*
 * The next 6 lines are an XBM bitmap file that has been
 * included into our code.
 */
#define weave_width 16
#define weave_height 16
static char weave_bits[] = {
   0x00, 0x1c, 0x00, 0x1c, 0xff, 0x9c, 0xff, 0x9c, 0xff, 0x9c, 0x00,
   0x1c, 0x00, 0x1c, 0x1c, 0x1c, 0x1c, 0x00, 0x1c, 0x00, 0x9c, 0xff,
   0x9c, 0xff, 0x9c, 0xff, 0x1c, 0x00, 0x1c, 0x00, 0x1c, 0x1c};

static XiFillPattern weave_pattern;

weave_pattern = XiFillPatternCreate(weave_bits,
                                     weave_width, weave_height,
                                     NULL, NULL);
```

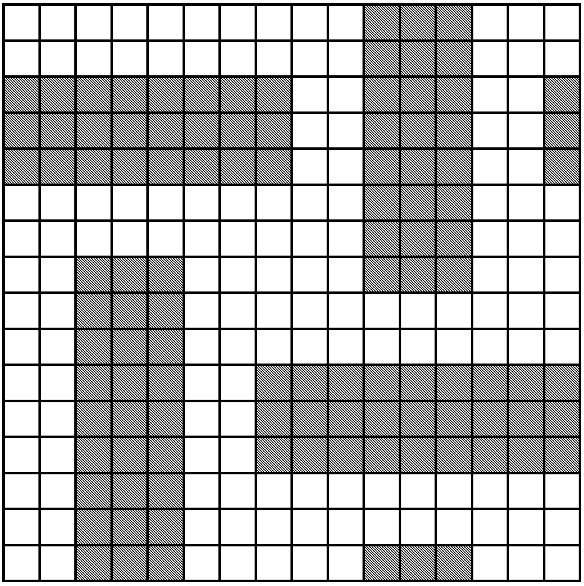


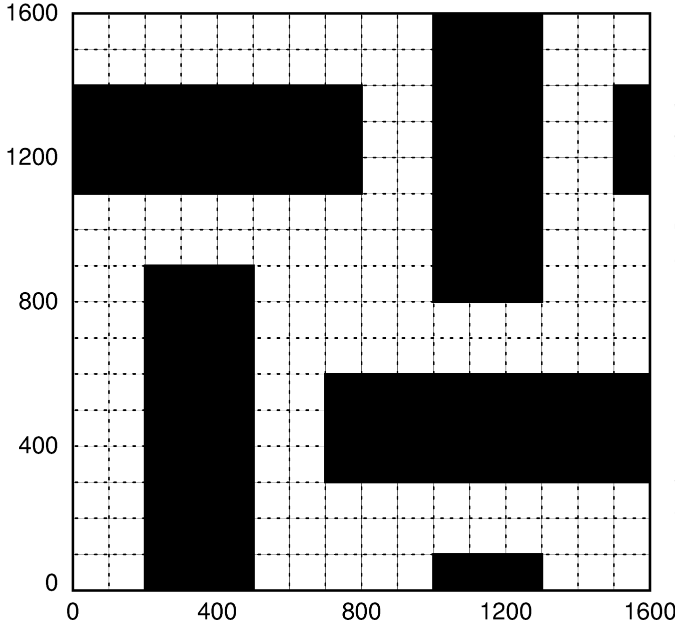
Figure 18-2. A bitmap fill pattern

The XiFillPattern created in this example will not work for any plot that is to be printed. The next section explains how to define a PostScript fill pattern.

18.1.2 Defining a Fill Pattern for PostScript

Fill patterns are handled in PostScript in the same basic way that they are handled in X--a single copy (or "cell") of the pattern is repeated over and over to tile the area to be filled. Since PostScript is a resolution independent graphics language, however, you cannot simply define a bitmap for the fill pattern--it would come out at a different size on different printers. Instead, you must write PostScript commands to draw a single cell of the pattern. The XiFillPattern abstraction will define a special font glyph with these PostScript commands, and will automatically handle the tiling operation.

As when defining a custom marker, you define the cell of a fill pattern using a special "high-resolution" coordinate system. Figure 18-3 shows this coordinate system for a single cell of the pattern, and has our example pattern drawn into it.



```
350 0 moveto
350 900 lineto
700 450 moveto
1600 450  lineto
0 1250 moveto
800 1250 lineto
1150 1600 moveto
1150 800 lineto
1150 0 moveto
1150 100 lineto
1500 1250 moveto
1600 1250 lineto
300 setlinewidth
stroke
```

Figure 18-3. A PostScript fill pattern and its coordinate system

As with the coordinate system for custom markers, 100 units in these coordinates are approximately equal to the size of one pixel on an X display, and dimensions of the PostScript cell are 100 times as large as the dimensions of the X bitmap that you specified with the second and third arguments to XiFillPatternCreate(). As with custom markers, and all PostScript code, the coordinates on the Y axis increase upwards, unlike coordinates in X which increase downwards. Unlike the coordinate system for custom markers, however, this one has its origin at the lower left corner of the cell, rather than in the middle of the cell.

Figure 18-3 also shows the PostScript code required to draw this cell of the fill pattern. Note that we set the line width to 300, but that this does not result in extremely wide lines, because line width is measured in the same coordinate system in which all the lines are drawn. Recall that in PostScript the coordinates given for a line are the centerpoint of the line, and half of the line width extends on either side. So the first line drawn in this example is centered on the X coordinate 350 and extends for 150 units to either side--from 200 to 500. (By default, it is only the "sides" of the lines that extend for 150 units on either side; the "end" or "caps" of the lines are flush against the endpoints. This is controlled by the setlinecap operator.) To understand this PostScript, you must remember that the lineto operator does not actually draw lines; it simply stores away lines to be drawn later by the stroke operator. This is why we can set the line width after defining the lines to be drawn.

Using some PostScript abbreviations supplied by the Plotter library, we can translate this fill pattern description to the following string.   (You can find the standard set of XiPlotter abbreviations at the top of any XiPlotter PostScript file, or listed in Chapter 17, *Defining Custom Markers*.)

```
static String weave_ps = "\
350 0 M 350 900 L \
700 450 M 1600 450 L \
0 1250 M 800 1250 L \
1150 1600 M 1150 800 L \
1150 0 M 1150 100 L \
1500 1250 M 1600 1250 L \
300 IL ST";
```

This is the PostScript code used to define the custom fill patterns used in the graphs pictured on the opening page of this chapter. "M" and "L" are obviously abbreviations for moveto and lineto. "IL" is an abbreviation for "initialize line". It sets the line width to the specified value, and also initializes all the other line attributes (line pattern, join style, and cap style) to their defaults--fill patterns are implemented as font glyphs, and this line attribute initialization is a required step for font glyph descriptions which is omitted from the PostScript code shown in the figure. Finally, "ST" is an abbreviation for stroke, which actually draws the lines.

The XiPlotter set of PostScript abbreviations also includes "m" and "l", which are alternate abbreviations for moveto and lineto. These m and l operators transform their coordinates to device coordinates and then back again to the current transform. This results in a rounding off, so that lines fall evenly onto device coordinates. If your fill pattern has a number of evenly spaced lines that are close together, you may find, using M and L that the lines are not spaced perfectly evenly--even at 300 dpi, this effect is noticeable with some patterns. If you switch to using m and l, your lines will be evenly spaced, but the trade-off will be that they may no longer have exactly the same width as one another. You may want to try out both sets of operators--some patterns look better with the first scheme, some look better with the other.

Now that we have defined a PostScript string for a single cell of our fill pattern, we can use it as the fourth argument to XiFillPatternCreate() to create an XiFillPattern that works with both X and PostScript. This is shown in Example 18-2.

Example 18-2. Creating an XiFillPattern for X and PostScript

```
/* the X version of the pattern */
#include "weave.xbm"

/* The PostScript version of the pattern */
static String weave_ps = "\
350 0 M 350 900 L \
700 450 M 1600 450 L \
0 1250 M 800 1250 L \
1150 1600 M 1150 800 L \
1150 0 M 1150 100 L \
1500 1250 M 1600 1250 L \
300 IL ST";

static XiFillPattern weave_pattern;

weave_pattern = XiFillPatternCreate(weave_bits,
                                     weave_width, weave_height,
                                     weave_ps, NULL);
```

Recall, though, that XiFillPatternCreate takes five arguments. The fifth argument is an alternate method for defining a PostScript fill pattern, and is only used in special cases. If you specify a string as the fifth argument to XiFillPatternCreate(), then the Plotter library will fill the area using the standard PostScript fill operator. This fifth argument allows you to set attributes that will affect this fill operation. In practice, the only thing you are likely to use here is the setgray operator, which can be abbreviated as "SG". The predefined fill patterns XiFillPatternGray1 through XiFillPatternGray5 use this scheme instead of defining a cell to be tiled (it would be hard to define a resolution-independent gray cell using lines.) You can use it when defining custom gray patterns--just use a string like ".75 SG" You should never specify a string for both the fourth and the fifth arguments, of course; pass NULL for one or the other.

18.2 Naming and Looking up Fill Patterns

Once you have created a custom fill pattern with XiFillPatternCreate(), you can use that marker directly in your C code, without any further action. You can also give the new marker a name, so that the String-to-XiFillPattern converter will recognize it and it can be used from a resource file. (Assuming, of course, that you create the pattern and register its name before you create any of the widgets that will be using it.) The XiFillPattern resource converter is described in Chapter 3, *Plotter Data Types and Resources.*

You can register a name for a marker you have created with the function XiFillPatternRegister() which takes two arguments: the marker name, and the XiFillPattern pointer.

```
void XiFillPatternRegister(String name, XiFillPattern pattern);
```

`name` The name under which the pattern is to be registered.

`pattern` The pattern that is being named.

```
XiFillPattern XiFillPatternLookup(String name);
```

`name` The name of the pattern that is to be looked up.

`Returns` The predefined or registered XiFillPattern with the specified name, or NULL if no such pattern is found.

Figure 18-4. XiFillPatternRegister() and XiFillPatternLookup()

You might register the pattern defined in the previous example as follows:

```
    XiFillPatternRegister("weave", weave_pattern);
```

Once a pattern is registered, the XiFillPattern resource converter will know about it. You can also look up registered fill patterns by name directly by calling XiFillPatternLookup() which takes a single string argument, and returns an XiFillPattern, if any is found by the specified name, or NULL if none is found.   You can use this function to look up the predefined patterns by name as well--it recognizes exactly the same names that the resource converter does.

Widget Resource Reference

This appendix gathers together the widget resource tables that have appeared in this manual for easy reference. The tables are organized alphabetically within this appendix, but the items within the tables are generally arranged by functional group, rather than alphabetically. If you need more information about a widget resource, the last column of the tables contains the page number at which you can find a detailed explanation. If you need to look something up, but do not know which table it would appear in or which function group within that table, try the index.

A.1 Annotation Resources

Name	Class	Type	Default	CSG	Pg
Position					
XmNxPercent	XmCXPercent	XmRInt	*Computed*	CSG	184
XmNyPercent	XmCYPercent	XmRInt	*Computed*	CSG	184
XmNxPixel	XmCXPixel	XmRPosition	0	CSG	185
XmNyPixel	XmCYPixel	XmRPosition	0	CSG	185
XmNxPoint	XmCXPoint	XmRDouble	*Computed*	CSG	185
XmNyPoint	XmCYPoint	XmRDouble	*Computed*	CSG	185
Offset					
XmNxPercentOffset	XmCPercentOffset	XmRInt	*Computed*	CSG	186
XmNyPercentOffset	XmCPercentOffset	XmRInt	*Computed*	CSG	186
XmNxPixelOffset	XmCPixelOffset	XmRPosition	0	CSG	185
XmNyPixelOffset	XmCPixelOffset	XmRPosition	0	CSG	185
XmNadjustOffset	XmCAdjustOffset	XmRBoolean	TRUE	CSG	186
Size and margins					
XmNmarginWidth	XmCMargin	XmRDimension	2	CSG	187
XmNmarginHeight	XmCMargin	XmRDimension	2	CSG	187
XmNboxWidth	XmCWidth	XmRDimension	*Computed*	CSG	187
XmNboxHeight	XmCHeight	XmRDimension	*Computed*	CSG	187
Colors					
XmNforeground	XmCForeground	XmRPixel	XmNforeground[a]	CSG	187
XmNbackground	XmCBackground	XmRPixel	XmNbackground[b]	CSG	187
Arrows					
XmNdrawArrow	XmCDrawArrow	XmRBoolean	TRUE	CSG	188
XmNarrowColor	XmCArrowColor	XmRPixel	XmNforeground[a]	CSG	189
XmNarrowShaftPattern	XmCArrowShaftPattern	XmRLinePattern	NULL	CSG	189
XmNarrowShaftWidth	XmCArrowShaftWidth	XmRDimension	2	CSG	189
XmNarrowHeadWidth	XmCArrowHeadSize	XmRDimension	6	CSG	189
XmNarrowHeadLength	XmCArrowHeadSize	XmRDimension	6	CSG	189
XmNarrowHeadDistance	XmCArrowDistance	XmRDimension	4	CSG	189
XmNarrowTailDistance	XmCArrowDistance	XmRDimension	2	CSG	189
Shadows and Trim					
XmNdrawShadow	XmCDrawShadow	XmRBoolean	FALSE	CSG	189
XmNdrawArrowShadow	XmCDrawShadow	XmRBoolean	FALSE	CSG	190
XmNshadowColor	XmCForeground	XmRPixel	XmNforeground[a]	CSG	190
XmNstippleShadow	XmCStippleShadow	XmRBoolean	TRUE	CSG	190
XmNshadowXOffset	XmCShadowOffset	XmRPosition	4	CSG	190
XmNshadowYOffset	XmCShadowOffset	XmRPosition	4	CSG	190
XmNtrimColor	XmCForeground	XmRPixel	XmNforeground[a]	CSG	190
XmNtrimWidth	XmCTrimWidth	XmRDimension	0	CSG	190
TextPlot Resources					
XmNfontList	XmCFontList	XmFontList	XmNfontList[d]	CSG	191
XmNpsFontList	XmCPSFontList	XiPSFontList	XmNpsFontList[e]	CSG	191
XmNtextString	XmCTextString	XmString	NULL[c]	CSG	191
XmNtext	XmCText	XmRString	NULL	CSG	191
XmNfontSize	XmCFontSize	XmRDimension	12	CSG	191

XmNfontStyle	XmCFontStyle	XmRString	NULL	CSG	191
XmNalignment	XmCAlignment	XmRAlignment	XiALIGNMENT_ CENTER	CSG	192
XmNstringDirection	XmCStringDirection	StringDirection	*dynamic*	CSG	192
XmNopaque	XmCOpaque	XmRBoolean	TRUE	CSG	192
ImagePlot Resources					
XmNpixmap	XmCPixmap	XmRPixmap	None	CSG	192
XmNbitmap	XmCBitmap	XmRPixmap	None	CSG	192
XmNimage	XmCImage	XmRXImage	NULL	CSG	192
XmNclipMask	XmCClipMask	XmRPixmap	None	CSG	193

[a] *Inherited from the Plotter XmNforeground resource.*

[b] *Inherited from the Plotter XmNbackground resource.*

[c] *When an XmString resource is unset, its value is created from the corresponding String resource, using a font tag derived from the corresponding style and size resources.*

[d] *Inherited from the Plotter XmNfontList resource.*

[e] *Inherited from the Plotter XmNpsFontList resource.*

A.2 Axis Resources

Name	Class	Type	Default	CSG	Pg
Title					
XmNtitleFontList	XmCFontList	XmFontList	XmNfontList[a]	CSG	82
XmNtitlePSFontList	XmCPSFontList	XiPSFontList	XmNpsFontList[b]	CSG	82
XmNtitleString	XmCTitleString	XmString	NULL[c]	CSG	82
XmNtitle	XmCTitle	String	NULL	CSG	83
XmNtitleSize	XmCFontSize	Dimension	14	CSG	83
XmNtitleStyle	XmCFontStyle	String	NULL	CSG	84
XmNtitleColor	XmCForeground	Pixel	XmNforeground[d]	CSG	84
Labels					
XmNdrawLabels	XmCDrawLabels	Boolean	TRUE	CSG	84
XmNlabelFontList	XmCFontList	XmFontList	XmNfontList[a]	CSG	84
XmNlabelPSFontList	XmCPSFontList	XiPSFontList	XmNpsFontList[b]	CSG	84
XmNlabelSize	XmCFontSize	Dimension	10	CSG	85
XmNlabelStyle	XmCFontStyle	String	NULL	CSG	85
XmNlabelColor	XmCForeground	Pixel	XmNforeground[d]	CSG	85
Axis and Tick Marks					
XmNlineColor	XmCForeground	Pixel	XmNforeground[d]	CSG	85
XmNlineWidth	XmCLineWidth	Dimension	0	CSG	85
XmNmarkWidth	XmCLineWidth	Dimension	0	CSG	86
XmNmarksInside	XmCMarksInside	Boolean	FALSE	CSG	86
XmNmarksOutside	XmCMarksOutside	Boolean	TRUE	CSG	86
XmNmarkLength	XmCMarkLength	Dimension	5	CSG	86
XmNsubmarkLength	XmCMarkLength	Dimension	2	CSG	86
Frame and Origin					
XmNdrawFrame	XmCDrawFrame	Boolean	TRUE	CSG	87
XmNdrawOrigin	XmCDrawOrigin	Boolean	TRUE	CSG	87
XmNoriginWidth	XmCLineWidth	Dimension	0	CSG	87
XmNoriginColor	XmCForeground	Pixel	XmNforeground[d]	CSG	87
Grid					
XmNdrawGrid	XmCDrawGrid	Boolean	TRUE	CSG	88
XmNgridColor	XmCForeground	Pixel	XmNforeground[d]	CSG	88
XmNgridWidth	XmCLineWidth	Dimension	0	CSG	88
XmNgridPattern	XmCGridPattern	LinePattern	"\001"	CSG	88
XmNdrawSubgrid	XmCDrawSubgrid	Boolean	FALSE	CSG	89
XmNsubgridColor	XmCForeground	Pixel	XmNforeground[d]	CSG	89
XmNsubgridWidth	XmCLineWidth	Dimension	0	CSG	89
XmNsubgridPattern	XmCGridPattern	LinePattern	"\003"	CSG	89
Layout					
XmNmargin	XmCMargin	Dimension	2	CSG	89
XmNaxisWidth	XmCAxisWidth	Dimension	0	CSG	89
Auto Scaling[e]					
XmNautoScaleMin	XmCAutoScale	Boolean	TRUE	CSG	91

XmNautoScaleMax	XmCAutoScale	Boolean	TRUE	CSG	92
XmNautoScaleMargin	XmCAutoScaleMargin	Dimension	5	CSG	92
Bounds and Transform[e]					
XmNmax	XmCMax	Double	10.00	CSG	93
XmNmin	XmCMin	Double	0.00	CSG	93
XmNunitMultiplier	XmCUnitMultiplier	Double	1.00	CSG	94
XmNunitConstant	XmCUnitConstant	Double	o.0	CSG	94
XmNlog	XmCLog	Boolean	FALSE	CSG	96
Auto Marking[e]					
XmNautoMark	XmCAutoMark	Boolean	TRUE	CSG	97
XmNroundEndpoints	XmCRoundEndpoints	Boolean	TRUE	CSG	97
XmNpixelsPerLabel	XmCPixelsPerLabel	Dimension	50	CSG	98
XmNpixelsPerMark	XmCPixelsPerMark	Dimension	10	CSG	98
Manual Axis Marking[e]					
XmNlabelInterval	XmCLabelInterval	Short	1	CSG	100
XmNlabelOffset	XmCLabelOffset	Short	0	CSG	100
XmNmarkValues	XmCMarkValues	DoubleList	NULL	CSG	100
XmNnumMarks	XmCNumMarks	Cardinal	0	CSG	100
XmNmarkMultiplier	XmCMarkMultiplier	Double	1.0	CSG	100
XmNmarkConstant	XmCMarkConstant	Double	0.0	CSG	100
XmNmarkValueType	XmCMarkValueType	PlotDataType	XiPlotDataDouble	CSG	101
XmNmarkRecordSize	XmCMarkRecordSize	Cardinal	sizeof(double)	CSG	101
XmNmarkRecordOffset	XmCMarkRecordOffset	Cardinal	0	CSG	101
Auto Labeling[e]					
XmNautoLabel	XmCAutoLabel	Boolean	TRUE	CSG	101
XmNlabelFormat	XmCLabelFormat	String	"%g"	CSG	101
XmNlabelFunction	XmCLabelFunction	AxisLabelProc	NULL	CSG	101
Manual Axis Labeling[e]					
XmNlabels	XmCLabels	XmStringTable	NULL	CSG	105
XmNnumLabels	XmCNumLabels	Dimension	0	CSG	105
XmNlabelRecordSize	XmCLabelRecordSize	Cardinal	sizeof(String)	CSG	105
XmNlabelRecordOffset	XmCLabelRecordOffset	Cardinal	0	CSG	105

[a] *Inherited from the XmNfontList resource*

[b] *Inherited from the XmNpsFontList resource*

[c] *When an XmString resource is unset, its value is created from the corresponding String resource using a font tag derived from the corresponding style and size resources.*

[d] *Inherited from the Plotter SmNforeground resource*

[e] *Do not set these resources when using a DBPak subclass of this widget.*

A.3 BarPlot Resources

Name	Class	Type	Default	CSG	Pg
FillPattern					
XmNfillColor	XmCForeground	Pixel	XmNforeground[a]	CSG	137
XmNfillBackground	XmCBackground	Pixel	XmNplotAreaColor[b]	CSG	137
XmNfillPattern	XmCFillPattern	FillPattern	XiFillPatternSolid	CSG	137
XmNfillOpaque	XmCFillOpaque	Boolean	TRUE	CSG	137
Lines					
XmNlineColor	XmCForeground	Pixel	XmNforeground[a]	CSG	139
XmNlineWidth	XmCLineWidth	Dimension	0	CSG	139
Shadows					
XmNdrawShadow	XmCDrawShadow	Boolean	FALSE	CSG	139
XmNshadowColor	XmCForeground	Pixel	XmNforeground[a]	CSG	139
XmNshadowPattern	XmCShadowPattern	FillPattern	XiFillPatternGray3	CSG	139
XmNshadowXOffset	XmCShadowOffset	Position	4	CSG	139
XmNshadowYOffset	XmCShadowOffset	Position	4	CSG	139
Bars					
XmNbarSize	XmCBarSize	Short	80	CSG	142
XmNbarPosition	XmCBarPosition	Short	0	CSG	142
XmNstacked	XmCStacked	Boolean	FALSE	CG	142
XmNorigin	XmCOrigin	Double	0.0	CSG	142
Data[c]					
XmNnumValues	XmCNumValues	Cardinal	0	CSG	143
XmNxMin	XmCXMin	Double	1.0	CSG	143
XmNxInterval	XmCXInterval	Double	1.0	CSG	143
XmNvalues	XmCValues	Pointer	NULL	CSG	143
XmNvalueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	143
XmNvalueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	143
XmNvalueOffset	XmCValueOffset	Cardinal	0	CSG	143
XmNvalueConstant	XmCValueConstant	Double	0.0	CSG	143
XmNvalueMultiplier	XmCValueMultiplier	Double	1.0	CSG	143

[a] *Inherited from the Plotter XmNforeground resource.*

[b] *Inherited from the Plotter XmNplotAreaColor resource.*

[c] *Do not set these resources when using a DBPak subclass of this widget.*

A.4 CallbackPlot Resources

Name	Class	Type	Default	CSG	Pg
Untyped Data					
XmNdata0	XmCData	XiUntyped	NULL	CSG	213
XmNdata1	XmCData	XiUntyped	NULL	CSG	213
XmNdata2	XmCData	XiUntyped	NULL	CSG	213
XmNdata3	XmCData	XiUntyped	NULL	CSG	213
XmNdata4	XmCData	XiUntyped	NULL	CSG	213
XmNdata5	XmCData	XiUntyped	NULL	CSG	213
XmNdata6	XmCData	XiUntyped	NULL	CSG	213
XmNdata7	XmCData	XiUntyped	NULL	CSG	213
XmNdata8	XmCData	XiUntyped	NULL	CSG	213
XmNdata9	XmCData	XiUntyped	NULL	CSG	213
Callbacks					
XmNchangeCallback	XmCCallback	Callback	NULL	C	217
XmNcreateCallback	XmCCallback	Callback	NULL	C	217
XmNdrawCallback	XmCCallback	Callback	NULL	C	217
XmNdrawPSCallback	XmCCallback	Callback	NULL	C	217
XmNfreeCallback	XmCCallback	Callback	NULL	C	218
XmNlegendCallback	XmCCallback	Callback	NULL	C	218
XmNlegendPSCallback	XmCCallback	Callback	NULL	C	219
XmNresizeCallback	XmCCallback	Callback	NULL	C	219
XmNshadowCallback	XmCCallback	Callback	NULL	C	219
XmNshadowPSCallback	XmCCallback	Callback	NULL	C	219
Control Resources					
XmNrecalc	XmCRecalc	Boolean	FALSE	S	220
XmNredraw	XmCRedraw	Boolean	FALSE	S	220
XmNxMax	XmCXMax	double	-DBL_MAX	CSG	220
XmNxMin	XmCXMin	double	DBL_MAX	CSG	220
XmNyMax	XmCYMax	double	-DBL_MAX	CSG	220
XmNyMin	XmCYMin	double	DBL_MAX	CSG	220

A.5 ErrorPlot Resources

Name	Class	Type	Default	CSG	Pg
Error Bars					
XmNerrorBarColor	XmCForeground	Pixel	XmNforeground[a]	CSG	127
XmNerrorBarWidth	XmCLineWidth	Dimension	0	CSG	127
XmNerrorCapLength	XmCErrorCapLength	Dimension	3	CSG	127
Fixed Errors					
XmNfixedError	XmCFixedError	Double	0.0	CSG	128
XmNpercentError	XmCPercentError	Double	0.0	CSG	128
Variable Errors[b]					
XmNnumErrorValues	XmCNumValues	Cardinal	0	CSG	129
XmNhighValues	XmCValues	Pointer	NULL	CSG	129
XmNhighValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	129
XmNhighValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	129
XmNhighValueOffset	XmCValueOffset	Cardinal	0	CSG	129
XmNlowValues	XmCValues	Pointer	NULL	CSG	129
XmNlowValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	129
XmNlowValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	129
XmNlowValueOffset	XmCValueOffset	Cardinal	0	CSG	129

[a] *Inherited from the Plotter XmNforeground resource.*

[b] *Do not set these resources when using a DBPak subclass of this widget.*

A.6 FadePlot Resources

Name	Class	Type	Default	CSG	Pg
Colors					
XmNcolors	XmCColors	XmColorList	NULL	CSG	202
XmNstartColor	XmCStartColor	Pixel	XmNplotAreaColor[a]	CSG	202
XmNendColor	XmCEndColor	Pixel	XmNplotAreaColor[a]	CSG	202
XmNnumColors	XmCNumColors	Dimension	10	CSG	203
XmNfadeDirection	XmCFadeDirection	XiFadeDirection	top_to_bottom	CSG	203
Color Interpolation					
XmNuseHSL	XmCUseHSL	Boolean	FALSE	CSG	203
XmNinterpolateFunc	XmCInterpolateFunc	XiInterpolateFunc	NULL	CSG	203
Printing					
XmNdontPrint	XmCDontPrint	Boolean	FALSE	CSG	204
XmNpsNumColors	XmCNumColors	Dimension	100	CSG	204

[a] *Inherited from the Plotter XmNplotAreaColor resource.*

A.7 HighLowPlot Resources

Name	Class	Type	Default	CSG	Pg
Bars					
XmNbarColor	XmCForeground	Pixel	XmNforeground[a]	CSG	171
XmNbarWidth	XmCLineWidth	Dimension	5	CSG	171
XmNlineColor	XmCForeground	Pixel	XmNforeground[a]	CSG	171
XmNlineWidth	XmCLineWidth	Dimension	3	CSG	171
XmNlineLength	XmCLineLength	Dimension	6	CSG	171
Data[b]					
XmNnumValues	XmCNumValues	Cardinal	0	CSG	171
XmNxValues	XmCXValues	Pointer	NULL	CSG	171
XmNxValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	171
XmNxValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	171
XmNxValueOffset	XmCValueOffset	Cardinal	0	CSG	171
XmNxValueConstant	XmCValueConstant	Double	0.0	CSG	171
XmNxValueMultiplier	XmCValueMultiplier	Double	1.0	CSG	171
XmNhighValues	XmCValues	Pointer	NULL	CSG	171
XmNhighValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	171
XmNhighValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	171
XmNhighValueOffset	XmCValueOffset	Cardinal	0	CSG	171
XmNhighValueConstant	XmCValueConstant	Double	0.0	CSG	171
XmNhighValueMultiplier	XmCValueMultiplier		Double	CSG	171
XmNlowValues	XmCValues	Pointer	NULL	CSG	171
XmNlowValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	171
XmNlowValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	171
XmNlowValueOffset	XmCValueOffset	Cardinal	0	CSG	171
XmNlowValueConstant	XmCValueConstant	Double	0.0	CSG	171
XmNlowValueMultiplier	XmCValueMultiplier	Double	1.0	CSG	171
XmNcloseValues	XmCValues	Pointer	NULL	CSG	171
XmNcloseValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	171
XmNcloseValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	171
XmNcloseValueOffset	XmCValueOffset	Cardinal	0	CSG	171
XmNcloseValueConstant	XmCValueConstant	Double	0.0	CSG	171
XmNcloseValueMultiplier	XmCValueMultiplier	Double	1.0	CSG	171

[a] *Inherited from the Plotter XmNforeground resource.*

[b] *Do not set these resources when using a DBPak subclass of this widget.*

A.8 HistoPlot Resources

Name	Class	Type	Default	CSG	Pg
Bounds					
XmNmin	XmCMin	Double	0.0	CSG	149
XmNmax	XmCMax	Double	100.0	CSG	149
XmNcomputeBounds	XmCComputeBounds	Boolean	FALSE	CSG	149
Bins					
XmNnumBins	XmCNumBins	Short	-1[a]	CSG	150
XmNbinSize	XmCBinSize	Double	10.0	CSG	150
XmNbarSize[b]	XmCBarSize	Short	100	CSG	
Data[c]					
XmNnumRawValues	XmCNumValues	Cardinal	0	CSG	150
XmNrawValues	XmCValues	Pointer	NULL	CSG	150
XmNrawValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	150
XmNrawValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	150
XmNrawValueOffset	XmCValueOffset	Cardinal	0	CSG	150
XmNrawValueConstant	XmCValueConstant	Double	0	CSG	150
XmNrawValueMultiplier	XmCValueMultiplier	Double	1.0	CSG	150

[a] *An unspecified value; will be computed based on XmNbinSize.*

[b] *This is a BarPlot resource; HistoPlot changes the default value.*

[c] *Do not set these resources when using a DBPak subclass of this widget.*

A.9 LinePlot Resources

Name	Class	Type	Default	CSG	Pg
Lines					
XmNconnectPoints	XmCConnectPoints	Boolean	TRUE	CSG	111
XmNlineColor	XmCForeground	Pixel	XmNforeground[a]	CSG	111
XmNsegmentWidth	XmCLineWidth	Dimension	0	CSG	
XmNlinePattern	XmCLinePattern	LinePattern	XiLinePatternSolid	CSG	111
Markers					
XmNmarkPoints	XmCMarkPoints	Boolean	FALSE	CSG	112
XmNmarker	XmCMarker	Marker	XiMarkerCircle	CSG	112
XmNmarkerColor	XmCForeground	Pixel	XmNforeground[a]	CSG	113
XmNmarkerBackground	XmCBackground	Pixel	XmNplotAreaColor[b]	CSG	113
FillPattern					
XmNfillPattern	XmCFillPattern	FillPattern	XiFillPatternNone	CSG	114
XmNfillColor	XmCForeground	Pixel	XmNforeground[a]	CSG	114
XmNfillBackground	XmCBackground	Pixel	XmNplotAreaColor[b]	CSG	114
XmNfillOpaque	XmCFillOpaque	Boolean	TRUE	CSG	114
Impulses					
XmNdrawImpulses	XmCDrawImpulses	Boolean	FALSE	CSG	115
Shadows					
XmNdrawShadow	XmCDrawShadow	Boolean	FALSE	CSG	115
XmNshadowColor	XmCForeground	Pixel	XmNforeground[a]	CSG	115
XmNstippleShadow	XmCStippleShadow	Boolean	TRUE	CSG	116
XmNshadowXOffset	XmCShadowOffset	Position	2	CSG	116
XmNshadowYOffset	XmCShadowOffset	Position	2	CSG	116
Data[c]					
XmNnumValues	XmCNumValues	Cardinal	0	CSG	116
XmNxValues	XmCXValues	Pointer	NULL	CSG	116
XmNxValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	116
XmNxValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	116
XmNxValueOffset	XmCValueOffset	Cardinal	0	CSG	116
XmNxValueConstant	XmCValueConstant	Double	0.0	CSG	116
XmNxValueMultiplier	XmCValueMultiplier	Double	1.0	CSG	116
XmNyValues	XmCYValues	Pointer	NULL	CSG	116
XmNyValueType	XmCValueType	PlotDataType	XiPlotDataDouble	CSG	116
XmNyValueSize	XmCValueSize	Cardinal	sizeof(double)	CSG	116
XmNyValueOffset	XmCValueOffset	Cardinal	0	CSG	116
XmNyValueConstant	XmCValueConstant	Double	0.0	CSG	116
XmNyValueMultiplier	XmCValueMultiplier	Double	1.0	CSG	116

[a] *Inherited from the Plotter XmNforeground resource.*

[b] *Inherited form the Plotter XmNplotAreaColor resource.*

[c] *Do not set these resources when using a DBPak subclass of this widget.*

A.10 PiePlot Resources

Name	Class	Type	Default	CSG	Pg
Fill Pattern					
XmNfillColor	XmCForeground	Pixel	XmNforeground[a]	CSG	159
XmNfillBackground	XmCBackground	Pixel	XmNbackground[b]	CSG	159
XmNfillPattern	XmCFillPattern	FillPattern	XiFillPatternSolid	CSG	159
XmNfillOpaque	XmCFillOpaque	Boolean	TRUE	CSG	159
Lines					
XmNlineColor	XmCForeground	Pixel	XmNforeground[a]	CSG	160
XmNlineWidth	XmCLineWidth	Dimension	0	CSG	160
XmNdrawRadius	XmCDrawRadius	Boolean	TRUE	CSG	160
Labels					
XmNlabelFontList	XmCFontList	XmFontList	XmNfontList[c]	CSG	161
XmNlabelPSFontListf	XmCPSFontList	XiPSFontList	XmNpsFontList[d]	CSG	161
XmNlabelString	XmCLabelString	XmString	NULL[e]	CSG	162
XmNlabel	XmCLabel	String	NULL	CSG	162
XmNlabelSize	XmCFontSize	Dimension	12	CSG	162
XmNlabelStyle	XmCFontStyle	String	NULL	CSG	162
XmNlabelColor	XmCForeground	Pixel	XmNforeground[a]	CSG	163
XmNlabelRadius	XmCLabelRadius	Dimension	38	CSG	163
XmNshowValue	XmCShowValue	Boolean	FALSE	CSG	163
Shadows					
XmNdrawShadow	XmCDrawShadow	Boolean	FALSE	CSG	160
XmNshadowColor	XmCForeground	Pixel	XmNforeground[a]	CSG	160
XmNshadowPattern	XmCShadowPattern	FillPattern	XiFillPatternSolid	CSG	161
XmNshadowOpaque	XmCShadowOpaque	Boolean	TRUE	CSG	161
XmNshadowXOffset	XmCShadowOffset	Position	4	CSG	161
XmNshadowYOffset	XmCShadowOffset	Position	4	CSG	161
Layout					
XmNxPosition	XmCXPosition	Dimension	50	CG	164
XmNyPosition	XmCYPosition	Dimension	50	CG	164
XmNradius	XmCRadius	Dimension	35	CG	164
XmNexplodeRadius	XmCExplodeRadius	Dimension	0	CSG	165
XmNstartAngle[f]	XmCStartAngle	Position	90	CG	165
Data					
XmNvalue[g]	XmCValue	Double	0.0	CSG	163

[a] *Inherited from the Plotter XmNforeground resource.*

[b] *Inherited from the Plotter XmNplotAreaColor resource.*

[c] *Inherited from the Plotter XmNfontList resource.*

[d] *Inherited from the Plotter XmNpsFontList*

[ed] *When an XmString resource is unset, its value is created from the corresponding String resource, using a font tag derived from the corresponding style and size resources.*

[f] *This resource is ignored except for the first wedge in a pie chart.*

[g] *Do not set this resource when using a DBPak subclass of this widget.*

A.11 Plotter Resources

Name	Class	Type	Default	CSG	Pg
Colors					
XmNforeground	XmCForeground	Pixel	black	CSG	56
XmNbackground	XmCBackground	Pixel	white	CSG	56
XmNplotAreaColor	XmCBackground	Pixel	XmNbackground[a]	CSG	56
XmNbackdrop	XmCBackdrop	Widget	NULL	SG	57
Fonts					
XmNfontList	XmCFontList	XmFontList	see below	CG	57
XmNtitleFontList	XmCFontList	XmFontList	XmNfontList[c]	CSG	58
XmNlegendTitleFontList	XmCFontList	XmFontList	XmNfontList[c]	CSG	58
XmNlegendFontList	XmCFontList	XmFontList	XmNfontList[c]	CSG	58
XmNpsFontList	XmCPSFontList	XiPSFontList	NULL	CG	58
XmNtitlePSFontList	XmCPSFontList	XiPSFontList	XmNpsFontList[d]	CG	58
XmNlegendTitlePSFontList	XmCPSFontList	XiPSFontList	XmNpsFontList[d]	CG	59
XmNlegendPSFontList	XmCPSFontList	XiPSFontList	XmNpsFontList[d]	CG	59
Axes					
XmNxAxis[e]	XmCXAxis	Widget	NULL	G	59
XmNyAxis[e]	XmCYAxis	Widget	NULL[f]	G	59
XmNyAxis2[e]	XmCYAxis	Widget	NULL	G	59
XmNsecondYAxis	XmCSecondYAxis	Boolean	FALSE	CSG	59
Title					
XmNshowTitle	XmCShowTitle	Boolean	TRUE	CSG	61
XmNtitleString	XmCTitleString	XmString	NULL	CSG	60
XmNtitle	XmCTitle	String	NULL	CSG	60
XmNtitleSize	XmCFontSize	Dimension	18	CSG	60
XmNtitleStyle	XmCFontStyle	String	NULL	CSG	60
XmNtitleColor	XmCForeground	Pixel	XmNforeground[b]	CSG	61
Legend					
XmNshowLegend	XmCShowLegend	Boolean	TRUE	CSG	61
XmNlegendTitleString	XmCLegendTitleString	XmString	NULL[f]	CSG	62
XmNlegendTitle	XmCLegendTitle	String	"Legend"	CSG	62
XmNlegendTitleSize	XmCFontSize	Dimension	14	CSG	63
XmNlegendTitleStyle	XmCFontStyle	String	NULL	CSG	63
XmNlegendTitleColor	XmCForeground	Pixel	XmNforeground[b]	CSG	63
XmNlegendSize	XmCFontSize	Dimension	12	CSG	63
XmNlegendStyle	XmCFontStyle	String	NULL	CSG	63
XmNlegendColor	XmCForeground	Pixel	XmNforeground[b]	CSG	63
XmNlegendBackground	XmCBackground	Pixel	XmNbackground[a]	CSG	63
XmNlegendSelectForeground	XmCBackground	Pixel	XmNbackground[a]	CSG	63
XmNlegendSelectBackground	XmCForeground	Pixel	XmNforeground[b]	CSG	64
XmNlegendSpacing	XmCMargin	Dimension	4	CSG	64

XmNlegendIconWidth	XmCLegendIconWidth	Dimension	20	CSG	64
XmNlegendUnderline	XmCLegendUnderline	Boolean	TRUE	CSG	64
XmNlegendBox	XmCLegendBox	Boolean	TRUE	CSG	64
XmNlegendBoxWidth	XmCLineWidth	Dimension	2	CSG	64
XmNlegendBoxColor	XmCForeground	Pixel	XmNforeground[b]	CSG	64
Layout					
XmNmarginWidth	XmCMargin	Dimension	10	CSG	65
XmNmarginHeight	XmCMargin	Dimension	10	CSG	65
XmNtitleMinHeight	XmCTitleMinHeight	Dimension	5	CSG	65
XmNlegendMinWidth	XmCLegendMinWidth	Dimension	5	CSG	65
XmNlegendMargin	XmCLegendMargin	Dimension	10	CSG	65
Double Buffering					
XmNdoubleBuffer	XmCDoubleBuffer	Boolean	FALSE	CSG	66
XmNdoubleBufferFrequencyHint	XmCDoubleBufferFrequencyHint	DoubleBufferFrequencyHint	Frequent	CSG	66
XmNbackingStore	XmCBackingStore	BackingStore	FALSE	CSG	67
User Interaction					
XmNallowCrosshair	XmCAllowCrosshair	Boolean	TRUE	CSG	234
XmNcrosshairLineWidth	XmCLineWidth	Dimension	0	CSG	234
XmNcrosshairColor	XmCForeground	Pixel	XmNforeground[b]	CSG	234
XmNpermanentCrosshair	XmCPermanentCrosshair	Boolean	FALSE	CSG	235
XmNcrosshairMotionOnly	XmCCrosshairMotionOnly	Boolean	FALSE	CSG	
XmNallowDrag	XmCAllowDrag	Boolean	TRUE	CSG	238
XmNdragThreshold	XmCDragThreshold	Dimension	4	CSG	231
XmNdragLineWidth	XmCLineWidth	Dimension	0	CSG	238
XmNdragColor	XmCForeground	Pixel	XmNforeground[b]	CSG	238
XmNallowPan	XmCAllowPan	Boolean	TRUE	CSG	241
XmNpanLineWidth	XmCLineWidth	Dimension	0	CSG	241
XmNpanColor	XmCForeground	Pixel	XmNforeground[b]	CSG	241
XmNallowMultipleSelect	XmCAllowMultipleSelect	Boolean	TRUE	CSG	229
Callbacks					
XmNactivateCallback	XmCCallback	Callback	NULL	C	229
XmNclickCallback	XmCCallback	Callback	NULL	C	231
XmNdoubleClickCallback	XmCCallback	Callback	NULL	C	231
XmNdragCallback	XmCCallback	Callback	NULL	C	237
XmNdragMotionCallback	XmCCallback	Callback	NULL	C	237
XmNeditCallback	XmCCallback	Callback	NULL	C	243
XmNleaveCallback	XmCCallback	Callback	NULL	C	234
XmNmotionCallback	XmCCallback	Callback	NULL	C	233

XmNpanCallback	XmCCallback	Callback	NULL	C	240
XmNpanMotionCallback	XmCCallback	Callback	NULL	C	240
XmNselectCallback	XmCCallback	Callback	NULL	C	228
Cursors					
XmNplotterCursor	XmCCursor	ColoredCursor	None	CSG	68
XmNclickCursor	XmCCursor	ColoredCursor	crosshair	CSG	68
XmNpanCursor	XmCCursor	ColoredCursor	fleur	CSG	241
XmNtopLeftCursor	XmCCursor	ColoredCursor	ul_angle	CSG	68
XmNtopRightCursor	XmCCursor	ColoredCursor	ur_angle	CSG	68
XmNbottomLeftCursor	XmCCursor	ColoredCursor	ll_angle	CSG	68
XmNbottomRightCursor	XmCCursor	ColoredCursor	lr_angle	CSG	68
Printing					
XmNforceColorPrinting	XmCForceColorPrinting	Boolean	FALSE	CSG	
XmNcustomPSPrefix	XmCCustomPSPrefix	String	NULL	CSG	69
XmNcustomPSSuffix	XmCCustomPSSuffix	String	NULL	CSG	69
Constraint Resources					
XmNbackgroundPlot	XmCBackgroundPlot	Boolean	FALSE	CSG	70
XmNlegendString	XmCLegendString	XmString	NULL[f]	CSG	70
XmNlegendName	XmCLegendName	String	NULL	CSG	70
XmNdrawingOrder	XmCDrawingOrder	Short	1	CSG	70
XmNselected	XmCSelected	Boolean	FALSE	CSG	71

[a] *Inherited from the XmNbackground resource.*

[b] *Inherited from the Xmforeground resource.*

[c] *Inherited from the XmNfontList resource.*

[d] *Inherited from the XmNpsFontList resource*

[e] *Plotter axes are implemented as independent Axis objects, each with a full set of its own resources.*

[f] *When an XmString resource is unset, its value is created from the corresponding String resource, using a font tag derived from the corresponding style and size resources.*

B

Function Reference

This appendix shows the signatures for all of the public functions in the Plotter library. They are organized alphabetically by widget or module, rather than by alphabetically by name.

B.1 Annotation Functions

```
Widget XiCreateTextPlot(Widget parent, String name,
                         ArgList args, Cardinal num_args);

Widget XiCreateImagePlot(Widget parent, String name,
                          ArgList args, Cardinal num_args);

void XiAnnotationPosition(Widget annotation, double x, double y,
                           int xoffset, int yoffset)

Boolean XiLinePlotGetPoint(Widget w, int point,
                            double *x_return, double *y_return)

Boolean XiHighLowPlotGetPoint(Widget w, int point,
                               double *x_return, double *high_return,
                             double *low_return, double *close_return)

Boolean XiBarPlotGetBar(Widget w, int bar,
                         double *x0_return, double *y0_return,
                         double *x1_return, double *y1_return)

void XiPiePlotGetPosition(Widget pie,
           double *x0_return, double *y0_return, /* vertex */
           double *x1_return, double *y1_return, /* start of arc */
          double *x2_return, double *y2_return, /* end point of arc */
          double *x3_return, double *y3_return, /* mid-point of arc */
          double *percent_return)              /* wedge percentage */

void XiImagePlotUpdate(Widget w);
```

B.2 Axis Functions

```
Widget XiPlotterXAxis(Widget plotter);

Widget XiPlotterYAxis(Widget plotter);

Widget XiPlotterYAxis2(Widget plotter);

void XiAxisAutoscale(Widget w);

void XiAxisSetBounds(Widget w, double min, double max);

void XiAxisGetBounds(Widget w, double *min, double *max);

void XiAxisSetMarks(Widget w, double min, double max,
                     double mark_interval, double first_mark,
                     int label_interval, label_offset);

void XiAxisSetLabels(Widget w, double min, double max,
                      double mark_interval, double first_mark,
                      int label_interval, label_offset,
                      String *labels, Cardinal num_labels);

void XiAxisSetLabelStrings(Widget w, double min, double max,
                            double mark_interval, double first_mark,
                            int label_interval, label_offset,
                            XmString *label_strings, Cardinal
                            num_labels);

void XiConvertWorldToPixel(Widget plotter, double x, double y,
                            int *px_return, int *py_return);


  void XiConvertPixelToWorld(Widget plotter, int px, int py,
                              double *x_return, double *y_return);
```

B.3 BarPlot Functions

```
Widget XiCreateBarPlot(Widget parent, String name,
                        ArgList args, Cardinal num_args);

void XiBarPlotSetDoubleValues(Widget w, double xmin, double xinterval,
                               double *values, Cardinal num);

void XiBarPlotSetFloatValues(Widget w, double xmin, double xinterval,
                              float *values, Cardinal num);

void XiBarPlotSetIntValues(Widget w, double xmin, double xinterval,
                            int *values, Cardinal num);

void XiBarPlotSetValues(Widget w, double xmin, double xinterval,
                         XtPointer vals, XiPlotDataType type,
                        Cardinal size, Cardinal offset, Cardinal num);

void XiBarPlotDataChanged(Widget w);
```

B.4 CallbackPlot Functions

```
Widget XiCreateCallbackPlot(Widget parent, String name,
                             ArgList args, Cardinal num_args);
```

B.5 ErrorPlot Functions

```
Widget XiCreateErrorPlot(Widget parent, String name,
                          ArgList args, Cardinal num_args);

void XiErrorPlotSetDoubleErrors(Widget w, double *highvals, double
*lowvals,
                                 Cardinal num_values);

void XiErrorPlotSetFloatErrors(Widget w, float *highvals, float
*lowvals,
                                Cardinal num_values);

void XiErrorPlotSetIntErrors(Widget w, int *highvals, int *lowvals,
                              Cardinal num_values);

void XiErrorPlotSetErrors(Widget w,
                           XiPlotDataType type, Cardinal size,
                           XtPointer highvals, Cardinal highoffset,
                           XtPointer lowvals, Cardinal lowoffset,
                           Cardinal num_values);

void XiErrorPlotDataChanged(Widget w);
```

B.6 FadePlot Functions

```
Widget XiCreateFadePlot(Widget parent, String name,
                         ArgList args, Cardinal num_args);
```

B.7 FillPattern Functions

```
XiFillPattern XiFillPatternCreate(char *bits,
                                    int width, int height,
                                    String psglyph, String psfill);

void XiFillPatternRegister(String name, XiFillPattern pattern);

XiFillPattern XiFillPatternLookup(String name);
```

B.8 HighLowPlot Functions

```
Widget XiCreateHighLowPlot(Widget parent, String name,
                            ArgList args, Cardinal num_args);

void XiHighLowPlotSetDoubleValues(Widget w,
                                   double xmin, double xinterval,
                                   double *highvals, double *lowvals,
                                   double *closevals,
                                   Cardinal num_points);

void XiHighLowPlotSetFloatValues(Widget w,
                                  double xmin, double xinterval,
                                  float *highvals, float *lowvals,
                                  float *closevals,
                                  Cardinal num_points);

void XiHighLowPlotSetIntValues(Widget w,
                                double xmin, double xinterval,
                                int *highvals, int *lowvals,
                                int *closevals,
                                Cardinal num_points);

void XiHighLowPlotSetValues(Widget w, double xmin, double xinterval,
                            XiPlotDataType ytype, Cardinal ysize,
                            XtPointer highvals, Cardinal highoffset,
                            XtPointer lowvals, Cardinal lowoffset,
                           XtPointer closevals, Cardinal closeoffset,
                            Cardinal num_points);

void XiHighLowPlotSetPoints(Widget w,
                            XtPointer xvals, XiPlotDataType xtype,
                            Cardinal xsize, Cardinal xoffset,
                            XiPlotDataType ytype, Cardinal ysize,
                            XtPointer highvals, Cardinal highoffset,
                            XtPointer lowvals, Cardinal lowoffset,
                           XtPointer closevals, Cardinal closeoffset,
                            Cardinal num_points);

void XiHighLowPlotDataChanged(Widget w);
```

B.9 HistoPlot Functions

```
Widget XiCreateHistoPlot(Widget parent, String name,
                          ArgList args, Cardinal num_args);

void XiHistoPlotSetDoubleValues(Widget w,double *values,Cardinalnum);

void XiHistoPlotSetFloatValues(Widget w, float *values, Cardinal num);

void XiHistoPlotSetIntValues(Widget w, int *values, Cardinal num);

void XiHistoPlotSetValues(Widget w,
                           XtPointer vals, XiPlotDataType type,
                         Cardinal size, Cardinal offset,Cardinal num);

void XiHistoPlotDataChanged(Widget w);
```

B.10 LinePlot Functions

```
Widget XiCreateLinePlot(Widget parent, String name,
                         ArgList args, Cardinal num_args);

void XiLinePlotSetDoublePoints(Widget w,
                          double *xvals, double *yvals, Cardinal num);

void XiLinePlotSetFloatPoints(Widget w,
                            float *xvals, float *yvals, Cardinal num);

void XiLinePlotSetIntPoints(Widget w,int *xvals,int *yvals,Cardinal
                             num);

void XiLinePlotSetDoubleYValues(Widget w, double xmin, double
                              xinterval, double *yvals, Cardinal num);

void XiLinePlotSetFloatYValues(Widget w, double xmin, double
                               xinterval, float *yvals, Cardinal num);

void XiLinePlotSetIntYValues(Widget w, double xmin, double xinterval,
                              int *yvals, Cardinal num);

void XiLinePlotSetPoints(Widget w, XtPointer xvals, XiPlotDataType
xtype,
                          Cardinal xsize, Cardinal xoffset,
                          XtPointer yvals, XiPlotDataType ytype,
                          Cardinal ysize, Cardinal yoffset, Cardinal
                          num);

void XiLinePlotSetYValues(Widget w, double xmin, double xinterval,
                           XtPointer yvals, XiPlotDataType ytype,
                           Cardinal ysize, Cardinal yoffset, Cardinal
                           num);

void XiLinePlotDataChanged(Widget w);

void XiLinePlotDataAdded(Widget w, Cardinal new_num);
```

B.11 Marker Functions

```
XiMarker XiMarkerCreate(int width, int height,
                          char *mark_bits, char *mask_bits,
                          String psmark, String psmask);


void XiMarkerRegister(String name, XiMarker marker);

XiMarker XiMarkerLookup(String name);
```

B.12 PiePlot Functions

```
Widget XiCreatePiePlot(Widget parent, String name,
                        ArgList args, Cardinal num_args);
```

B.13 Plotter Functions

```
Widget XiCreatePlotter(Widget parent, String name,
                        ArgList args, Cardinal num_args);

Widget XiPlotterXAxis(Widget plotter);

Widget XiPlotterYAxis(Widget plotter);

Widget XiPlotterYAxis2(Widget plotter);

void XiPlotterDisableRedisplay(Widget w);

void XiPlotterEnableRedisplay(Widget w);

void XiPlotterRelayout(Widget plotter);

void XiPlotterPrint(Widget plotter, char *filename,
                     int width, int height,
                     double scale, int landscape);
```

C

Data Type Reference

This appendix lists the definitions of the public data structures used by the Plotter library, and contains tables of predefined markers, fill patterns, and line patterns.

C.1 Callback Structures

```
typedef struct {
    int reason;        /* the type of the callback */
    XEvent *event;     /* the event that triggered it */
    Position px, py;   /* pixel coordinates of the mouse */
    double x, y;       /* plot coordinates of the mouse */
} XiClickCallbackStruct, XiDoubleClickCallbackStruct,XiMotionCall-
backStruct;

typedef struct {
    int reason;        /* the type of the callback */
    XEvent *event;     /* the event that triggered it */
    Position px1, py1; /* pixel coords of upper left */
    Position px2, py2; /* pixel coords of lower right */
    double x1, y1;     /* plot coords of upper left */
    double x2, y2;     /* plot coords of upper left */
} XiDragCallbackStruct, XiDragMotionCallbackStruct;

typedef struct {
    int reason;        /* the type of the callback */
    XEvent *event;     /* the event that triggered it */
    Position pdx, pdy; /* delta x and delta y in pixel coords */
    double dx, dy;     /* delta x and delta y in plot coords */
} XiPanCallbackStruct, XiPanMotionCallbackStruct;

typedef struct {
    int reason;        /* the type of the callback */
    XEvent *event;     /* the event that triggered it */
    Boolean selected;  /* True if an item was selected; False if
                          de-selected */
    Widget item;       /* the item that was selected, if any */
    Widget *selected_plots;    /* a complete list of selected plots */
    Cardinal num_selected_plots;/* # of elements in the list */
} XiSelectCallbackStruct, XiActivateCallbackStruct;

typedef struct {
    int reason;
    XEvent *event;
} XiLeaveCallbackStruct;
.bp
#define XiPlotterEditTitle 1
#define XiPlotterEditXAxis 2
#define XiPlotterEditYAxis 3
#define XiPlotterEditYAxis2 4

typedef struct {
    int reason;
    XEvent *event;
    int where;             /* one of the above constants */
    Boolean double_click;  /* True if this was a double-click */
} XiEditCallbackStruct;
```

C.2 Callback Reasons

```
#define XmCR_CLICK         1 /* button clicked in plotting area */
#define XmCR_DOUBLE_CLICK  2 /* double click in plotting area */
#define XmCR_DRAG          3 /* end of drag in plotting area */
#define XmCR_DRAG_MOTION   4 /* motion during drag in plotting area */
#define XmCR_PAN           5 /* end of pan in plotting area */
#define XmCR_PAN_MOTION    6 /* motion during pan in plotting area */
#define XmCR_SELECT        7 /* selection state changed in legend */
#define XmCR_ACTIVATE      8 /* double-click in legend */
#define XmCR_MOTION        9 /* all motion events */
#define XmCR_LEAVE        10 /* pointer has left plotting area */
#define XmCR_EDIT         11 /* button clicked on title or axes */
```

C.3 Data Types

```
typedef enum {
    XiPlotDataDouble,
    XiPlotDataFloat,
    XiPlotDataUnsignedLong,
    XiPlotDataLong,
    XiPlotDataUnsignedInt,
    XiPlotDataInt,
    XiPlotDataUnsignedShort,
    XiPlotDataShort,
    XiPlotDataUnsignedChar
} XiPlotDataType;

typedef struct {
    XtPointer values;
    Cardinal num;
    Cardinal size;
    Cardinal offset;
    unsigned char type;
    double constant;
    double multiplier;
} XiPlotData;
```

C.4 Fill Patterns

LeftHatch	RightHatch	Crosshatch
XiFillPatternLefthatch1 XiFillPatternLefthatch2 XiFillPatternLefthatch3 XiFillPatternLefthatch4 XiFillPatternLefthatch5	XiFillPatternRighthatch1 XiFillPatternRighthatch2 XiFillPatternRighthatch3 XiFillPatternRighthatch4 XiFillPatternRighthatch5	XiFillPatternCrosshatch1 XiFillPatternCrosshatch2 XiFillPatternCrosshatch3 XiFillPatternCrosshatch4 XiFillPatternCrosshatch5
Grid	**Gray**	**Special**
XiFillPatternGrid1 XiFillPatternGrid2 XiFillPatternGrid3 XiFillPatternGrid4 XiFillPatternGrid5	XiFillPatternGray1 XiFillPatternGray2 XiFillPatternGray3 XiFillPatternGray4 XiFillPatternGray5	XiFillPatternNone XiFillPatternSolid

A String-to-XiFillPattern converter is automatically registered by the BarPlot and PiePlot widgets. It recognizes the names shown in the table, with the "XiFillPattern" prefix removed, and all letters converted to lowercase. It is case-sensitive.

C.5 Line Patterns

Predefined Line Patterns		
C	**Resource File**	**Pattern**
XiLinePatternSolid	Solid	none
XiLinePatternDotted1	Dotted1	1 2
XiLinePatternDotted2	Dotted2	1 4
XiLinePatternDotted3	Dotted3	1 6
XiLinePatternDashed1	Dashed1	3 3
XiLinePatternDashed2	Dashed2	5 5
XiLinePatternDashed3	Dashed3	7 7
XiLinePatternDotDashed1	DotDashed1	3 1 1 1
XiLinePatternDotDashed2	DotDashed2	5 2 1 2
XiLinePatternDotDashed3	DotDashed3	7 3 1 3

A String-to-LinePattern converter is registered by the Plotter widget. It recognizes the pattern names shown in the table and is case-sensitive. This converter can also convert custom line patterns specified as a whitespace-separated list of integers (as shown in the third column of the table).

C.6 Markers

Tiny Markers	**Small Markers**
XiMarkerTinySquare	XiMarkerSmallSquare
XiMarkerTinyFilledSquare	XiMarkerSmallFilledSquare
XiMarkerTinyCircle	XiMarkerSmallCircle
XiMarkerTinyFilledCircle	XiMarkerSmallFilledCircle
XiMarkerTinyTriangle	XiMarkerSmallTriangle
XiMarkerTinyFilledTriangle	XiMarkerSmallFilledTriangle
XiMarkerTinyTriangle2	XiMarkerSmallTriangle2
XiMarkerTinyFilledTriangle2	XiMarkerSmallFilledTriangle2
XiMarkerTinyDiamond	XiMarkerSmallDiamond
XiMarkerTinyFilledDiamond	XiMarkerSmallFilledDiamond
XiMarkerTinyPlus	XiMarkerSmallPlus
XiMarkerTinyX	XiMarkerSmallX
Normal Markers	**Large Markers**
XiMarkerSquare	XiMarkerLargeSquare
XiMarkerFilledSquare	XiMarkerLargeFilledSquare
XiMarkerCircle	XiMarkerLargeCircle
XiMarkerFilledCircle	XiMarkerLargeFilledCircle
XiMarkerTriangle	XiMarkerLargeTriangle
XiMarkerFilledTriangle	XiMarkerLargeFilledTriangle
XiMarkerTriangle2	XiMarkerLargeTriangle2
XiMarkerFilledTriangle2	XiMarkerLargeFilledTriangle2
XiMarkerDiamond	XiMarkerLargeDiamond
XiMarkerFilledDiamond	XiMarkerLargeFilledDiamond
XiMarkerPlus	XiMarkerLargePlus
XiMarkerX	XiMarkerLargeX

A String-to-XiMarker resource converter is registered by the LinePlot widget. It recognizes the names shown in the table, with the "XiMarker" prefix removed, and capitalization retained. It is case-sensitive.