An X11 Graphics Extension for the ROSE Database System

by

David T. Loffredo

A Project Submitted to the Graduate Faculty of
Rensselaer Polytechnic Institute
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE

Approved:

Dr. Martin Hardwick
Project Advisor

Rensselaer Polytechnic Institute
Troy, New York
May 1989
# Contents

## Project Report

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 History</td>
<td>1</td>
</tr>
<tr>
<td>1.3 The X Toolkit</td>
<td>2</td>
</tr>
<tr>
<td>1.4 Integration of Event Processing</td>
<td>3</td>
</tr>
<tr>
<td>1.5 Linking the Interface to Computation</td>
<td>4</td>
</tr>
<tr>
<td>1.5.1 Callbacks</td>
<td>5</td>
</tr>
<tr>
<td>1.5.2 Event handlers</td>
<td>6</td>
</tr>
<tr>
<td>1.5.3 Translation Tables</td>
<td>6</td>
</tr>
<tr>
<td>1.6 The X Library</td>
<td>6</td>
</tr>
<tr>
<td>1.7 System Implementation</td>
<td>7</td>
</tr>
<tr>
<td>1.7.1 The Rose Extension Mechanism</td>
<td>7</td>
</tr>
<tr>
<td>1.7.2 Modifications to Rose</td>
<td>9</td>
</tr>
<tr>
<td>1.7.3 Widget Table</td>
<td>9</td>
</tr>
<tr>
<td>1.7.4 Rose Software Support</td>
<td>11</td>
</tr>
<tr>
<td>1.8 Conclusion</td>
<td>12</td>
</tr>
</tbody>
</table>

## A File Descriptions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 C Source</td>
<td>15</td>
</tr>
<tr>
<td>A.1.1 Header Files</td>
<td>15</td>
</tr>
<tr>
<td>A.1.2 Internal Support</td>
<td>15</td>
</tr>
<tr>
<td>A.1.3 UNIX Based</td>
<td>15</td>
</tr>
<tr>
<td>A.1.4 Xlib Based</td>
<td>16</td>
</tr>
<tr>
<td>A.1.5 X Toolkit Based</td>
<td>16</td>
</tr>
</tbody>
</table>
Acknowledgements

The author is indebted to Alok Mehta for sharing both his ROSE expertise and sound advice, to David McIntyre and David Tonnesen for their support as fellow X-novitiate-named-Daves, and to Joe W. Ruffles and Steven Vingerhoet, who’s good cheer made the nocturnal hours less oppressive and more productive.

I would also like to extend my thanks to Martin Hardwick and Dave Spooner for their guidance on this project, and to Pam Paslow for her invaluable administrative assistance.

Finally, to the many people at the Design Research Center, the Department of Computer Science, and General Electric who have assisted me in untold ways, I would like to express my most sincere appreciation.

This work was supported in part by the Program for Data Engineering of the Rensselaer Design Research Center, the Rensselaer Department of Computer Science, and the General Electric Corporation. The views presented herein are those of the author and do not necessarily reflect those of Rensselaer Polytechnic Institute or the General Electric Corporation.

X is Copyright (c) 1984, 1985, 1986, 1987 by Massachusetts Institute of Technology. UNIX is a trademark of AT&T.
DEC, MicroVax, and UIS are trademarks of Digital Equipment Corporation.
Sun, SunCore, and NeWs are trademarks of Sun Microsystems.
HP is a trademark of Hewlett Packard Company.
1.1 Introduction

This chapter describes the design of X/Rose, a portable graphics subsystem for the Rose programming language, with facilities intended for the production of graphic user interfaces and basic two dimensional graphics. This chapter examines the various issues and constraints that shaped the final X/Rose system as well as the rational behind the design decisions.

1.2 History

In the past, the graphics features of Rose have been largely machine dependent. Separate implementations existed for the Sun and MicroVax workstations. The Sun version used the SunCore package, while the MicroVax version used DEC’s User Interface Services (UIS) package. Each version supported different features, making system specific Rose code a necessity.

At the inception of this project, the X window system seemed to be a natural choice for a graphics platform. It has recently been gaining wide acceptance as the windowing and basic graphics standard for workstations. Implementations exist for several operating systems and a wide range of hardware. In addition, extensions currently under development include a Phigs extension to X (PEX), a separate 3-D server enhancement (X3D), and combination of X and Sun’s NeWS system (X-NeWS). It appears that X will provide, at least for the near future, a suitable, system independent, graphics platform.

The project described herein officially started in September of 1987, with some experimental X10.4 code written by Lisa Pratico, a summer student. This work was an early attempt to directly convert some of the UIS implementation to X. Unfortunately, there is no direct correspondence between the two systems. UIS is a higher level user interface package with extended facilities for engineering graphics while the
Xlib C language interface is a very low level package designed to provide the kind of machine independent functionality upon which something like UIS might be built. Because of this fundamental difference between the two platforms, it became evident that a system based solely upon the Xlib would be much more complex than the comparable UIS system. It was not clear how to proceed.

1.3 The X Toolkit

The Second Annual X Technical Conference set the tone for future work. It became clear that the X toolkit (Xtk) was the way to go. The X toolkit is built on top of the Xlib, and provides an object oriented approach to user interface construction, as well as facilities to manage things like application startup, resource management, data conversion, memory allocation, and X event processing. [McCor87]

Most importantly, the toolkit provided the framework necessary to construct user interfaces at a high level, using object oriented concepts. Where the Xlib concerns itself with windows, the toolkit concerns itself with high level user interface abstractions called widgets.

A great deal of power is encapsulated within these widgets. This feature was of great importance due to the nature of this project. By endowing these widgets with a modicum of intelligence, we essentially divide the application into a C level user interface and a Rose level computational component. The volume of communication between the computational and interface components is reduced because the widgets can do some rudimentary processing of input. Thus, the resulting macrocommunication [Harts89] better represents the semantics of the application and the computational component is divorced from the lexical details of the interface. In addition, communication from Rose to C has a high overhead so excessive communication would result in significant performance degradation.

The toolkit provides a framework for building and using these widgets, but at the time of this work, the only public domain widget library was the Athena widget set [Swick87]. Consequently, the X/Rose system contains only these widget classes. Since that time, more widgets, such as the HP widget set [HP88], have entered the public domain. It was expected that more widgets would become available, so provisions have been made for the addition of new widget classes to X/Rose.

It is the one of the functions of the X server to monitor all windows, input, and output. When keyboard and mouse input are detected, the server associates the input with a window, and packages this data into communication packets called events. Events are also generated to announce such things as a change in window size or position, the destruction of a window’s contents, a change of input focus, or the change of the machine’s colormap. These events are then sent over the network connection to the
application which created the window. It is the responsibility of the application to respond to each event in an appropriate manner.

This mechanism imposes a particular organization upon X applications. Traditional Xlib applications are built around a switch statement within an infinite loop. Each pass through the loop removes an event from the server queue and, given the event type and window ID, uses the switch statement to determine what actions (if any) are needed. This single loop must be able to deal effectively with all possible events and must produce all application functionality. Since this means handling everything from window refresh to application semantics, the structure of this loop quickly becomes twisted and obstruse. The effort involved in understanding, modifying, or debugging nontrivial applications quickly becomes staggering.

The X toolkit addresses these problems by providing higher level (abstractions that simplify development of the user interface to an X application [Asente]. Whereas the primitive building blocks under Xlib were windows, the primitive units under the toolkit are user interface components called widgets. Visually, a widget and a window are identical. Indeed, a widget is a window, but it is more than that. A widget is a programming unit which encapsulates private data and methods in an object oriented fashion [Cox86]. A widget’s methods are invoked when X events are sent to the window associated with that widget. These methods provide varying levels of semantic feedback, perform housekeeping functions within the widget, and invoke application functionality.

1.4 Integration of Event Processing

The toolkit provides the underlying support for reading the X event stream and mapping events to appropriate widget methods. This mapping process involves examining the event and mapping the window identifier to the corresponding widget. When the widget is found, this mechanism examines the event type to determine if the widget has provided a method for such an event. If so, this method is invoked. Thus, in this framework, the widget writer need only consider methods appropriate to an individual widget and the limited set of events that widget may receive.

Because this event dispatching mechanism is out of sight, it is easy to imagine widgets as happily executing coprocesses. One forgets that it is the execution of this event dispatch mechanism, not simply the receipt of an event, that causes a widget’s methods to be invoked. If this dispatch mechanism is not called, server events will remain queued and all widgets will cease to function.

Rose is built around an interpretive loop much like the read-eval loop in lisp. Within this loop, the system can examine several command sources, and then evaluate any string that it finds. For an X/Rose application to function properly, the toolkit event
dispatch mechanism must be integrated into this evaluation loop. To see this more clearly, consider an X/Rose application. The computational component is implemented with Rose code while the user interface is built from toolkit widgets. For the application to function, the user interface must be driven by the toolkit dispatch mechanism, while the computational functions must be executed by the Rose eval procedure.

The Rose eval loop has several modes and, depending on the mode, will examine a different source of input. Integration of X event and Rose command processing is accomplished by adding a new mode to the eval loop. This mode examines an internal command queue, evaluates anything it finds, and then calls the toolkit dispatch mechanism. Unlike the other modes, this new mode does not block execution when trying to read new input. Thus, if no Rose command input is available, the processing of X events continues.

The only way a C function can invoke a Rose command is by adding an executable string to the rose command queue. If, during the processing of an X event, a widget must invoke a rose function, it does so by putting a command string in a rose command queue. The Rose command will not be executed until the processing of that X event is completed. In practice, this system works well. Both Rose commands and X events are processed with speed and ease, and neither processing mechanism requires major modification.

As an aside, there are other toolkits, such as InterViews [Linto89], which distribute event processing throughout the toolkit code. Events might be handled by a central mechanism but other sections of the toolkit remain free to set up local event processing loops. Integrating Rose and such a toolkit this would have been considerably more difficult and would probably require modifying large sections of Rose and toolkit code.

1.5 Linking the Interface to Computation

It was a straightforward task to integrate the Xtk functions to create and manipulate widget structures, but once it was possible to create these structures, it was necessary to find a way to link Rose code to widget actions. The X toolkit provides three ways to do this, event handlers, callback lists, and translation tables.

An event handler is the lowest level mechanism. The user registers a function, or event handler, that will be invoked by the widget upon the receipt of a specified event. The function is passed any any data contained in the event structure.

A callback is very similar to a event handler, but the invocation of the callback function is not necessarily tied to any particular X event. In fact callbacks might be thought of as event handlers for “synthetic” events, like the “synthetic” tokens in [Jacob86], which represent conditions of higher level semantics than those of the
primitive X events. Upon invocation, the callback function may be passed data appropriate to the callback semantics.

A translation table is a set of bindings, contained in a widget, that maps widget-defined actions to X events. Applications are also able to define actions. Bindings to these application defined actions may be added to any widget’s translation table. Conceptually, a translation table is a mapping from lexical primitives to semantic actions. The advantage of using these bindings is that a user may customize any widget’s behavior by simply specifying a new translation table in a defaults file. Thus, one may change behavior without modifying or recompiling any code.

1.5.1 Callbacks

Initially, it seemed most natural to tie rose code to widgets by using widget callbacks. The user would specify a widget, callback list, and a string containing an executable rose command. The string would be given an identifier and stored in the rose workspace. An appropriate C function would be added to the callback list of the widget, and would be passed the identifier of the above Rose command string.

Upon invocation, the callback function will construct the string representations of the command identifier and any relevant return values. From these, it will construct, and pass to the interpreter, a command string which, when evaluated, will retrieve the original command string from the Rose workspace, append the return values as parameters, and evaluate the result.

When a new widget class is added to X/Rose, the system programmer must specify a callback function for each callback list that the widget supports. These functions should be able to carry out the above command construction process for any values returned by the callback. The system already contains functions for returning the usual primitive data types, but if a callback returns an unusual value, it a simple matter to add a new function for it.
1.5.2 Event handlers

Callbacks are limited in that each widget only supports a certain set of callback lists. These callbacks represent specific widget semantics and may not satisfy every need. This limitation was addressed by allowing the programmer to link Rose code directly to X events. Unfortunately, doing this ties the computational component of the application to the lexical portion of the interface. It would be preferable to link the computational component to the higher level semantics, but when dealing with a limited widget set, this is not always possible. Nevertheless, the distinction between callback and event handler has been hidden by a layer of Rose code so as to provide a single, uniform framework for the X/Rose programmer.

1.5.3 Translation Tables

Another way to address the problem would be to define an application action that could be used in the translation table of any widget. This action would take, as a parameter, the command identifier mentioned above, and, when invoked, would construct a command string to be passed to the interpreter. The advantage of this method is that the user can modify the binding from X events to this special action without changing any application code.

Realistically though, this advantage may be slim, since this new action would be in no way linked to the semantics of any widget. As with event handlers, the computational component of the application becomes too closely tied to the lexical portion of the interface. In fact, this mechanism would be equivalent to event handlers, but without the benefit of integration with callbacks in a consistent framework. In addition, this new application action would be unable to pass back the specialized data that many callbacks do. For these reasons, this mechanism is not supported by X/Rose.

1.6 The X Library

The toolkit provides excellent facilities for constructing user interfaces and handling X events, but it does not attempt to provide any advanced support for graphics. While it was no great chore to add Xlib graphics primitives to the interface, using them effectively in an application turned out to be a difficult task. Particularly troublesome is the requirement that application must be able to refresh the contents of any window whenever called upon to do so. Traditionally, applications are not written with this sort of functionality.

A solution is to provide a toolkit widget that supports engineering-type graphics with things like a world coordinate space, transforms, display lists and segments, and auto refresh. The gfx widget was an early attempt at such a graphics widget, but never
progressed beyond initial investigations. Alok Metha’s GMS, however, successfully addresses these problems by implementing, among other things, a pseudo-widget built on top of the X/Rose package.

1.7 System Implementation

The following sections describe the implementation of the X/Rose extension. This can be divided into several parts. First are ports, the C functions compiled into the Rose executable and callable from a Rose application. Second are modifications made to the original Rose system so that it might better accommodate X. Third are the widgets, user interface building blocks for Rose applications. Finally, there are the Rose functions which build upon the above features, and provide the application programmer with a consistent and convenient programming environment.

1.7.1 The Rose Extension Mechanism

Occasionally, advanced Rose applications need functions to access the features of the operating system or other software packages. In many cases it is difficult to develop such functions using the existing Rose features. For this reason, a mechanism within Rose enables the application writer to develop such functions in a different language, yet still use them in a Rose application. These functions, or ports, can be invoked by a Rose application and will operate on Rose data, but are implemented in C [Hardw86]. An unfortunate, but unavoidable, consequence is that the Rose executable must be relinked in order to add, delete, or modify ports.

These ports are assigned an identifier, called a port number, and are compiled into the Rose executable. Before an application may use a particular port, it must use the Rose open utility to enable calls to, also referred to as opening, the port. This utility accepts a port number and Rose domain, then determines whether the port can accept the indicated data structure.

Once opened, the application may invoke a port function by using the Rose send utility. This utility accepts a port number and data, then converts the data to an internal representation and passes it to the port function. The domain of the parameter data must be the same as the domain provided when the port was opened.

This mechanism served well for two previous graphics extensions, and seemed adequate for a third. Yet, a few improvements were made for the sake of modularity, maintainability, and efficiency. Originally, the open and send utilities contained large switch statements. New ports would be made known to these utilities by adding appropriate code to each switch statement. This arrangement was adequate for small
numbers of ports. The planned X extension, however, would initially consist of many ports, and would require the addition of more as new widget sets were integrated.

With this in mind, the case mechanism was scrapped in favor of a jump table. New ports were required to provide two C functions. The first, prefixed with `draw_`, would produce the functionality desired of the port, while the second, prefixed with `start_`, would be a predicate, able to determine whether the port is able to accept data in a given Rose domain. The `draw_` function would be called by the `send` utility while the `start_` function would be used by the `open` utility.

It is the duty of the `send` utility to transform Rose data into a format more accessible to C functions. This is done by decomposing the data into primitive values. These values are then stored in three buffers, one for float data, another for integer and boolean data, and the third for string data. These buffers are then passed to the C function, which must interpret it accordingly. This data dissection is done in an orderly manner, so it is possible for the function to interpret this data properly, but not without some knowledge about the original domain structure. Without this a priori knowledge, a function could not make any sense of the data in these buffers.

Because it is impossible to infer a domain from dissected data, a domain check mechanism must be provided. This is the function of the `open` utility. A port’s `start_` function is used to enforce type constraints, where the structure of a domain is analyzed to see if it matches the structure expected by the `draw_` function. If the domain is acceptable, than any future calls to this port must pass parameter data in that domain.

These buffers are fixed in size, and it has happened that calls with large arguments have overflowed the available space. This is an unfortunate situation, because there is no way to dynamically resize the buffers. This limitation remains in the current system, but I have included a proposal (see appendix C) for a revised parameter passing mechanism which removes these limitations.

The ports defined by the X/Rose extension can be divided into four classes.

**Toolkit Ports** The toolkit ports are used to build and manipulate widget structures. Most have been derived from the Xtk intrinsics.

**Graphics Ports** The graphics ports are used to allocate and manipulate server resources and to perform graphics operations in the windows of widgets. These ports are derived from a selected subset of Xlib functions.

**Class Specific Ports** The class specific ports are functions implemented by individual classes.

**System Ports** These ports are not tied to the X Window System, but are included here for completeness. Most are UNIX system calls.
1.7.2 Modifications to Rose

Beyond the addition of new ports, the Rose executable had to undergo other modifications. Provisions had to be made for the event processing as described in the previous chapter. As mentioned, this required the modifications for a command queue and the Xtk event dispatch mechanism. In addition, functions had to be added which would open the network connection with X server and initialize both the X toolkit and the extension.

Display Connection

Utilities for opening the network connection and initializing the toolkit were provided by Xlib and Xtk functions. These functions, along with additional code to perform extension specific initializations, merely had to be added to the Rose startup procedures.

Command Queue

The command queue existed in earlier graphics extensions, but was distributed throughout sections of unrelated code. To improve code modularity, the command queue and its operations were removed encapsulated, and formalized before inclusion into the X/Rose extension.

Event Processing

Much of this process is taken care of by toolkit provided functions. Two functions are provided, \texttt{XtNextEvent} and \texttt{XtDispatchEvent}, for removing a new event from the event queue and for and invoking the widget actions, respectively. If the event queue is empty, \texttt{XtNextEvent} blocks execution. This is acceptable because, in the normal case, Rose command input will never need to be processed while the system is waiting for event input. By definition, if the system is awaiting new events, then all Rose command input has been processed and new input can only result from the processing of events.

1.7.3 Widget Table

The toolkit is a framework and set of utilities with which widgets may be built, but the widgets themselves are the products of a programmer’s imagination. Thus, to take advantage of this programming environment, the widgets available under X/Rose should not be limited to some prechosen set. Rather, X/Rose should mimic the toolkit and define a framework within which any widget could be used.
Unfortunately, widget classes must be linked into the executable. It is not possible to
dynamically link widgets to the system, so a different approach had to be taken. The
system was defined so that new widgets may be added to the system in a very modular
way. Thus, widgets may not be dynamically linked, but they can be compiled in with
ease and without major modification of code.

Information about the widget classes available within X/Rose is organized into an
extensible internal table. The object-oriented nature of the toolkit serves to encapsu-
late most class specific information, but the small amount that remains is stored
as a table entry.

This table entry contains two bits of information about a widget class. First is the
class identifier for a class. This is simply a pointer to a static C structure, and is used
needed to create new instances of the class. Second is a list of callbacks a widget can
support. The elements of this list contain two fields, a string and a function pointer.
The string is used by toolkit intrinsics to identify the callback list while the function
is invoked by toolkit when the callback is triggered. This function must be able to
interpret the values returned by the callback construct a proper representation of
them for transmission to the Rose application.

In addition to these callbacks, the application writer can request notification on X
events. Unlike callback lists, the X events are fixed in number and defined for all
widgets, so a single function is able to serve as a universal event handler. This handler
must be able to process all types of event and construct an appropriate return string
for each.

In addition, this universal event handler needs special logic for handling expose events.
When a region of a window should be refreshed, the X server will send an event, called
an expose event, to inform the application that this region must be redrawn. This
event contains the description of a rectangle, but if the region is nonrectangular, the
server divides the region into its component rectangles and sends a contiguous series
of events to the application.

The typical refresh procedure doesn’t require such fine detail. In fact, most just use
the event as a signal to refresh the whole window. Because of this, logic was added to
merge these contiguous events into a single event which would describe the smallest
bounding rectangle of the exposed region.

Performing this event compression is not a difficult task. The event handler was mod-
ified to save expose data between calls. The first expose event carries a count of the
number of contiguous expose events, so it is known how many times the handler will
be called on this string of expose events. On subsequent calls, the handler calculates
a new minimal bounding rectangle, and after the last expose event is processed, the
callback function is invoked.
1.7.4 Rose Software Support

Startup

From Rose, ports are invoked by using the send mechanism. This requires a knowledge of the port numbers and the domains appropriate for each. To make this more convenient, Rose functions have been written for each port, which not only provide better mnemonics than port numbers, but also convert data from a form more useful for the application programmer to a form usable by the port.

These functions are defined in the X/Rose startup files (start.ams and others) These files also define the domains used by the extension and open each port.

Application Notification

Most port functions simply format data, but the Rose functions provided for application notification are more complex than this. Two ports are used by the notification mechanism. The first port adds an event handler or callback function, while the second removes such functions. Above these ports are built several Rose function.

X/Rose applications do not distinguish between X events and widget callback lists, rather, they work in terms of notify conditions. These notify conditions represent the callbacks and events to which a widget may respond, but they do so with minimal reference to implementation details.

When an application program requests notification, a Rose structure is created containing the identifier and class of a widget, a notification condition, and an executable command string. The application is given an identifier for this data, and is free to start and stop, as well as modify, the notification described by this structure. The application operates in terms of the more abstract “notification”, while the access functions are responsible for the addition or deletion of any event handlers or callbacks.

System Constants

X makes use of a large number of integer constants. Subsequently, many of the X/Rose functions require integer codes as arguments. At the present time, the system accepts over 170 different codes. Clearly, the application programmer can not be expected to know each code. As with the ports, human factors make it necessary to introduce some sort of mnemonics for the application programmer. These mnemonics were implemented by storing the constants in a Rose object set, in effect, allowing one to define names for integer values. The X/Rose functions were then modified to interpret these named constants.
Similarly, as the number of port functions and constants were growing, so were the number of startup files. To eliminate hardcoded path references within these files, symbolic paths were used. Like the integer constants, path strings are stored in a small Rose object set, and are given symbolic names. These paths are defined in a single file and may be easily updated when the X/Rose system is moved to a new directory structure.

1.8 Conclusion

By July of 1988, most of the work was complete. X/Rose had been ported to Sun, DEC, and HP and Solborne machines. In testament to the portability of X, the X/Rose system ran without any major, and very few minor, modifications. The package has since enjoyed considerable internal use, and seems to be serving its original purpose well.
References


[HP88] Hewlett-Packard Company, Programming with the HP X Widgets, Included in the MIT X software distribution.


[Schei87] Scheifler, Robert W., X Window System Protocol, Massachusetts Institute of Technology, 1987

Appendix A

File Descriptions

The following are short descriptions of all of the files that comprise the X/Rose system.

A.1 C Source

A.1.1 Header Files

RxEvent.h ...... Definitions for event processing.
RxNotify.h ..... Definitions for signaling Rose applications.
RxRose.h ........ General header information for all X/Rose files.
RxWidget.h ..... Definitions for accessing the X/Rose master widget table.
start_draw.h .. Parameter specifications for all port functions.

A.1.2 Internal Support

RxClose.c ...... Graphics extension shutdown code.
RxCmdq.c ....... Command queue procedures.
RxOpen.c ...... Graphics extension startup code.
start_draw.c .. Contains the port dispatch functions and a table of all ports available within Rose .

A.1.3 UNIX Based

enter.c ........ Port definition to return an input string from the console.
system.c ....... Port definition to execute a UNIX command.
A.1.4 Xlib Based

RxArc.c ....... Port definition to draw arcs.
RxBell.c ....... Port definition to sound the console bell.
RxClearArea.c Port definition to clear an area.
RxClearWin.c Port definition to clear a window.
RxContigLine.c Port definition to draw contiguous lines.
RxCopyArea.c Port definition to copy an area of a window.
RxCopyGC.c Port definition to copy one graphics context into another.
RxCrteGC.c Port definition to create a new graphics context.
RxDstyBitmap.c Port definition to destroy an allocated bitmap.
RxDstyCursor.c Port definition to destroy an allocated cursor.
RxDstyFont.c Port definition to unload a font from the server.
RxFillArc.c Port definition to draw filled arcs.
RxFillPoly.c Port definition to draw filled polygons.
RxFillRect.c Port definition to draw filled rectangles.
RxGetFPath.c Port definition to return the current font search path.
RxGetGC.c Port definition to return a graphics context attribute value.
RxLine.c Port definition to draw non-contiguous lines.
RxListFont.c Port definition to return a list of available fonts.
RxLoadBitmap.c Port definition to create a new bitmap from a file.
RxLoadColor.c Port definition to load a new color definition.
RxLoadCursor.c Port definition to create a new cursor type.
RxLoadFont.c Port definition to load a new font.
RxPoint.c Port definition to set individual pixels.
RxRectangle.c Port definition to draw rectangles.
RxSetFPath.c Port definition to set the font search path.
RxSetGC.c Port definition to set graphics context attribute values.
RxText.c Port definition to draw text.
RxTextExtent.c Port definition to return the screen dimensions of a bit of text.

A.1.5 X Toolkit Based

RxAddBind.c Port definition to add new bindings to a widget’s translation table.
RxCrteWidget.c Port definition to create new widget instances.
RxDstyWidget.c Port definition to destroy widget instances.
RxEvent.c All X Event processing routines.
RxFlushReq.c Port definition to flush cached server requests.
RxGetValues.c Port definition to return widget attribute values.
RxGrab.c ........ Port definition to constrain all mouse events to a particular widget.
RxHide.c ........ Port definition to remove a normal widget from the screen.
RxNotify.c ....... Contains routines to add event handlers and callbacks to widgets, to perform application notification, and to interpret X event structures.
RxPopdown.c ...... Port definition to remove a popup widget from the screen.
RxPopup.c ........ Port definition to place a popup widget on the screen.
RxRealize.c ...... Port definition to initialize a widget structure and place it on the screen.
RxSetBind.c ...... Port definition to replace the bindings in a widget’s translation table.
RxSetDflts.c ... Port definition of a general mechanism to set X/Rose system defaults.
RxSetSensit.c . Port definition to toggle the sensitivity of some widgets to user input.
RxSetValues.c . Port definition of a general mechanism to set the values of widget attributes.
RxSync.c .......... Port definition to send all server requests and wait for any subsequent events.
RxUngrab.c ....... Port definition to allow normal processing of mouse related events.
RxUnhide.c ....... Port definition to place a widget back on the screen.
RxWidget.c ...... Contains the master widget table for the system and various access functions for it.

A.1.6 Widget Class Specific

RxDialog.c ....... Class specific operations for the Athena Dialog widget.
RxGfx.c ......... Class specific operations for the Gfx prototype widget.
RxScroll.c ....... Class specific operations for the Athena Scrollbar widget.

A.2 AMS Source

start.ams ...... Central driver file
system.ams .... Contains Rose function definitions corresponding to the UNIX based ports described above.
graphics.ams .. Contains Rose function definitions corresponding to the Xlib based ports described above.
toolkit.ams ... Contains Rose function definitions corresponding to the Toolkit based ports described above. Also contains defi-
nitions to allow proper operation of the notification mechanism.

**classes.ams** ... Data definitions to correspond with the master widget table found in `RxWidget.c`. Also contains Rose function definitions corresponding to the widget class specific ports described above.

**start_db.ams** ... Central domain definitions required by all above Rose functions.

**constants.ams** ... Defines a database of symbolic integer constants to be used as various arguments to the above Rose functions.
## Appendix B

## Port Descriptions

The following is a list of ports defined by the X/Rose extension. Each entry lists the function of the port and the Rose domain accepted by the port. For the most current definitions of these domains, the reader should consult the file `start_db.ams`.

### UNIX Based

<table>
<thead>
<tr>
<th>Port</th>
<th>Function</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>getstring</td>
<td>rx_noargs</td>
</tr>
<tr>
<td>1</td>
<td>system</td>
<td>rx_str</td>
</tr>
</tbody>
</table>

### Toolkit Based

<table>
<thead>
<tr>
<th>Port</th>
<th>Function</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>RxCreateWidget</td>
<td>rx_value</td>
</tr>
<tr>
<td>11</td>
<td>RxDestroyWidget</td>
<td>rx_ilst</td>
</tr>
<tr>
<td>15</td>
<td>RxRealizeTree</td>
<td>rx_ilst</td>
</tr>
<tr>
<td>16</td>
<td>RxHide</td>
<td>rx_ilst</td>
</tr>
<tr>
<td>17</td>
<td>RxUnhide</td>
<td>rx_ilst</td>
</tr>
<tr>
<td>18</td>
<td>RxPopup</td>
<td>rx_popup</td>
</tr>
<tr>
<td>19</td>
<td>RxPopdown</td>
<td>rx_ilst</td>
</tr>
<tr>
<td>20</td>
<td>RxSetDefaults</td>
<td>rx_value</td>
</tr>
<tr>
<td>21</td>
<td>RxSetValues</td>
<td>rx_value</td>
</tr>
<tr>
<td>22</td>
<td>RxSetBindings</td>
<td>rx_binding</td>
</tr>
<tr>
<td>23</td>
<td>RxAddBindings</td>
<td>rx_binding</td>
</tr>
<tr>
<td>24</td>
<td>RxSetSensitive</td>
<td>rx_widspec</td>
</tr>
<tr>
<td>25</td>
<td>RxGrab</td>
<td>rx_widspec</td>
</tr>
<tr>
<td>26</td>
<td>RxUngrab</td>
<td>rx_ilst</td>
</tr>
<tr>
<td>27</td>
<td>RxGetValues</td>
<td>rx_valspec</td>
</tr>
<tr>
<td>31</td>
<td>RxStartNotify</td>
<td>rx_utfy</td>
</tr>
<tr>
<td>32</td>
<td>RxStopNotify</td>
<td>rx_utfy</td>
</tr>
<tr>
<td>35</td>
<td>RxFlushReq</td>
<td>rx_noargs</td>
</tr>
<tr>
<td>36</td>
<td>RxSync</td>
<td>rx_noargs</td>
</tr>
</tbody>
</table>
Port 37: RxBell

**Xlib based**

Graphics Context Operations
- Port 45: RxCreateGC
  Domain: rx_gc_value
- Port 46: RxSetGC
  Domain: rx_gc_value
- Port 47: RxGetGC
  Domain: rx_ilst
- Port 48: RxCopyGC
  Domain: rx_ilst

Font Operations
- Port 50: RxLoadFont
  Domain: rx_slst
- Port 51: RxDestroyFont
  Domain: rx_ilst
- Port 52: RxListFont
  Domain: rx_slst
- Port 53: RxGetFontPath
  Domain: rx_noargs
- Port 54: RxSetFontPath
  Domain: rx_slst

Pixmap Operations
- Port 55: RxLoadBitmap
  Domain: rx_slst
- Port 56: RxDestroyBitmap
  Domain: rx_ilst

Cursor Operations
- Port 60: RxLoadCursor
  Domain: rx_ilst
- Port 61: RxDestroyCursor
  Domain: rx_ilst

Area Operations
- Port 70: RxClearWin
  Domain: rx_noargs
- Port 71: RxClearArea
  Domain: rx_clr_area
- Port 72: RxCopyArea
  Domain: rx_cpy_area

Text Operations
- Port 80: RxDrawText
  Domain: rx_text
- Port 81: RxTextExtent
  Domain: rx_slst

Line Primatives
- Port 90: RxDrawArc
  Domain: rx_iarc
- Port 91: RxDrawLine
  Domain: rx_ilin
- Port 92: RxDrawPoint
  Domain: rx_ipnt
- Port 93: RxDrawContigLine
  Domain: rx_ipnt
- Port 94: RxDrawRectangle
  Domain: rx_irect

Fill Primatives
Port 100: RxFillArc  Domain: rx_iarc
Port 101: RxFillPolygon Domain: rx_ipnt
Port 102: RxFillRectangle Domain: rx_irect

**Widget Class Specific**

Gfx widget class
Port 110: RxGfxLine  Domain: rx_rlin
Port 111: RxGfxPoint Domain: rx_rpnt
Port 112: RxGfxLocator Domain: rx_noargs

Dialog widget class
Port 120: RxDialogGetValueString Domain: rx ilst

Scrollbar widget class
Port 125: RxScrollbarSetThumb Domain: rx_set_thumb
Appendix C

Proposed Extension Mechanism Modifications

The current extension mechanism presents a port function with three data buffers of set size, and it is the responsibility of the Rose application not to exceed these static limits. In addition, the domains of data passed through this interface are limited to, at most, a single aggregation. Certainly, this arrangement benefits from simplicity and efficiency, but given the flexibility of Rose, it seems that this limitation need not be present.

In the current system, port functions are passive entities, automatically given parameter data by the invocation mechanism. I propose that these functions be made into active entities, which must explicitly request any parameter data from the invocation mechanism. This would be implemented in the following fashion.

Parameter data will still be passed via three buffers: an integer array, a float array, and a string offset array with an associated character block, but each buffer must be fetched from Rose in the following manner. On call to the draw functions, the send dispatcher will provide an argument descriptor. This descriptor will be a pointer to an argument information structure, similar to the information used by isend, but with a few additions. These descriptors will have the following format:

```c
typedef argdesc
{int assno /* current place within an association
  int *stuff /* other rose stuff
}

typedef parmstruc
{int *intbuf
```
int    intsiz   /* Number of ints returned
int    maxint   /* Maximum size of the array
float *fltbuf  /* Addr to place floats
int    fltsiz   /* Number of floats returned
int    maxflt   /* Maximum size of the array
int    *StrBuf  /* Addr to place str offsets w/i ChrBuf
int    StrSiz   /* Number if strings returned
int    MaxStr   /* Maximum #of strings
char   *ChrBuf  /* Addr to place converted strings
int    MaxChr   /* Max size of character buffer
}

typedef enum {
    ParmError    /* Error in the parameters */
    ParmCont     /* Data too large, more awaits */
    ParmAll      /* No more data available */
} ReturnCode;

When a function desires parameter data, it passes this descriptor, along with a pointer
to a parameter structure (see above) to the parameter retrieval function. This function
will load the parameter structure with the desired data and return a status code. If
there is too much data to be passed at one time, subsequent calls to this function
will result in more and more data from the argument descriptor. One feature of
note is the difference in parameters expected by the open and draw functions. The
open functions require a representation of domain structure while the draw functions
require dissected Rose data. One way of handling this is to add a tag field to the
argument descriptor. Using this, the FetchParam function could be generalized to
both formats.

Some of the fields in the parameter structure will be pointers to data buffers. A
well behaved fill routine would provide these buffers if the requester did not. Such a
routine would maintain some static buffers for this purpose. Since data can be passed
in several steps, these static buffers need not be very large.

This modified extension mechanism should contain functions corresponding to the
following.

**Port Send(argcount)** - Does what isend does now, but just parses out the argument
structure and calls the dispatcher.

**Port Open(argcount)** - Equivalent to the open command. Calls a programmer
supplied routine for type checking and records the supplied domain.
**Port Close**(*argcount*) - Close port service. This may come in handy should any of these ports require cleanups.

**Interface Clear** - Closes all ports and clears all registered domains.

**Interface Dispatch**(*inum, argdesc*) - Equivalent to the `send` operation. Looks up an port function and invokes it.

**Interface function**(*argdesc*) - Various port functions

**FetchParm**(*argdesc, parmstruc*) - Fills the provided structure with fresh parameter data.

A final improvement is to allow multiple Rose arguments be passed to a port function. Within this proposed framework, such a modification could be easily incorporated. A list of argument descriptors, one for each Rose argument, would be passed to the a port function and it would fall upon this function to interpret the semantics properly. The only limitation to this arrangement is that, as specified, the FetchParm function would not be able to provide more than a single default buffer set to an application. (Unless we start allocating runtime storage, not a pleasant thing for what is supposed to be a quick parameter pass!) This enhancement would make it possible to pass some actual application data on a port open call, and would remove the need to compute enormous cartesian products for some port send calls. In addition, interface functions that require no parameters will see some speed improvement. Currently, every port send must take an argument, so parameterless procedures simply provide dummy arguments.
Appendix D

X/ROSE Users Guide
Appendix E

Introduction

The following manual describes X/Rose, the X window system extension to the Rose programming language. This extension provides a portable graphics subsystem for Rose, with facilities tailored for the production of graphic user interfaces and basic two-dimensional graphics. This manual is intended as both a reference guide for X/Rose, and an introduction to the X Window System. A basic understanding of Rose and a familiarity with windowing systems is assumed.

Chapter 2 of this manual is an overview of the X window system. Though not intended to turn the reader into an Xpert, this chapter will explain the various components of the X Window System and the roles they play, as well as the underlying strategies behind things like window management and optimization.

Chapter 3 introduces some of the basic concepts and terminology used by X and X/Rose. The reader is familiarized with widget instances and classes, class-specific resources and operations, and what these might be used for. In addition, the chapter discusses semantic events, notification, and the graphics features of X and the X/Rose extension.

Chapter 4 is an introduction to the use of this extension. This chapter should give the reader sufficient information to begin construction of an X/Rose application. Most X/Rose applications are found to have several phases and a specific overall organization. Each of these phases are briefly examined, and the overall organization of an application is explained. Two sample applications are used to highlight and further demonstrate this organization.

Chapter 5 is a command reference. Each X/Rose command is individually detailed, with exact parameter specifications as well as scenarios for use, warnings, and incidental information. In many cases, small examples have been provided.

Chapter 6 is a widget reference. Every widget class that may be used by an application is individually detailed, including descriptions of behavior, semantics, applicable resources, class-specific operations, and bindings (if any).
Appendix F

An Overview of X

“The X Window System is a network transparent windowing system that was designed at MIT and that runs under 4.3BSD UNIX, ULTRIX-32, many other UNIX variants, VAX/VMS, as well as several other operating systems”

– Various X Documentation

Since the above summary of X is so prevalent, (and so cryptic to the neophyte) I thought it an appropriate starting point for this overview.

The assertion that X is a windowing system implies that it allows one to divide a bitmap screen into windows, rectangular regions that may be written to, manipulated, and monitored much like abstract devices. Visually, these windows are allowed to overlap, and a window may contain another in a recursive manner. Text and graphic output operations may be performed in a window. These graphics operations are rather simple and based only on such 2-D primatives as lines, arcs, text, and bitmaps. More complex 3-D features like surfaces and shading have been left to future extensions.

The X Window System is based on client-server model. Normal application programs, called clients, that wish to display their data or receive input from the user, must do so by communicating with a special application program called a display server.

The claim to network transparency refers to this communication between client and server. Any communication situation requires that both parties must respect some common communication protocol. In this case, client and server use a specially defined communication format called the X protocol. This protocol has been designed to be usable over any reliable byte stream between the two parties. This means that the client and server can communicate via an IPC channel, a network connection, or even an RS-232 line. (although speed becomes a factor in the latter)
The full import of this last statement may not be apparent. Because the client and server can communicate via network connection, the client need not be on the same machine as the server nor even the same architecture. Issues such as architecture, location, and implementation language become immaterial, as long as the client and server can communicate via this protocol.

It is the function of this display server to perform all output to the bitmapped screen and to manage all input from the keyboard and mouse of a particular machine. Thus it makes sense for only one server to be running on a machine at any one time. This server must understand the X protocol, must be able to process graphics output requests and must be able to detect all user input. The server provides a link to a hardware, and will undoubtably require special device drivers and other hardware dependent code to perform these operations.

![Figure 1: The Client/Server Model](image)

Potentially many clients will be connected to a server at one time. This server must perform all output requests submitted by, and must route reports of user actions back to, these clients. Note that it is the server that encapsulates all hardware dependencies. The client is only required to be able to understand the protocol.

## F.1 The Layers of X

What we call “X” is really several layers of system. At the bottom layer is the X protocol. This protocol allows client and server to communicate so that operations may be requested and results returned. On top of this are language bindings. These are layers of code that hide the details of the communication protocol and allow one to perform this communication in a procedural fashion from a particular programming language.

Finally, on top of the X library is the X Toolkit. This is a framework that hides many of the lower level details and allows the manipulation of higher level constructs.
These constructs, called *widgets* are used as basic building blocks for graphic user interfaces.

The X/Rose graphics subsystem has been implemented on top of the X toolkit. One goal of this project has been to include as much of the versatility of X while managing as many of the system details as possible.

Figure 2: The many layers of X.

### F.2 The X Protocol

The X protocol two types of communication packet. Packets from the client to the server are used to request output operations and are called simply *requests*. Packets from the server to the client are used to notify the client of various happenings and are called *events*.

Requests are variable length packets consisting of an operation code and some associated data. The protocol defines the set of operation codes that must be recognized by the server. Events are fixed in length and also contain a identifying code and some data.

To reduce network overhead, many clients exploit the variable length structure of requests by grouping several similar output requests into a single large request. Similarly, clients are not notified in a synchronous fashion of user actions and the results of operations. Instead these *events* are sent back asynchronously and are placed in a queue for the client to process at its own leisure.

### F.3 The X Library

The X library is a set of C functions that automate the generation of X protocol requests. This library contains a wide variety of functions and features. It acts as the interface layer and takes care of such low level tasks as constructing and interpreting the communication packets between a client and the server. In addition,
these functions try to reduce the load on this connection by collecting similar graphics requests and sending them to the server at one time.

**F.4 The X Toolkit**

Even with the X library, developing applications is still a wearisome task. The X toolkit provides assistance in the form of a framework for developing user interfaces at a high level, using object oriented concepts.

Instead of dealing with windows, a mind boggling array of data values, and enormous case statements, the toolkit allows applications to be developed using user interface abstractions called *widgets*. A widget is a C construct encapsulating a window, private data, and code into one easily manipulated object. These widgets can be organized into a class hierarchy, with inheritance of methods and data. The basic widget class structure and all manipulation procedures have been packaged into what is called the *Toolkit Intrinsics*.

In addition, the toolkit contains a *Resource Manager*. This acts as a database and translation tool to manage the wide range of default and user configurable data values that are available to widgets.

Along with the toolkit intrinsics and resource manager, the X toolkit package contains a sample set of widgets, known as the *Athena Widget Set*. These widgets provide basic user interface functionality, but are by no means the only ones available.

**F.5 Policy**

X has been implemented to be a policy free a possible. It has been left to applications such as window managers to provide things like title bars, icons, and window manipulation commands. In addition, governing philosophies for such things as window arrangement, (*stacking vs. tiling*) and keyboard focus (*real estate vs. click-to-type*) can be determined by the window manager.

Window managers are handled differently in X than in other windowing systems. Other systems often place the window manager within the server. Under X, however, a window manager is just another client program. Several different styles of manager are available and the user is able to choose the one that best suits his needs.
Appendix G

The Graphics Environment

This chapter introduces some of the basic concepts and terminology used by X and X/Rose. Widget instances and classes, class-specific resources and operations will be explained as well as what these might be used for. In addition, this chapter will discuss semantic events, notification, and some of the graphics features of X and the X/Rose extension.

G.1 Widgets

G.1.1 Description

Widgets are the fundamental building blocks of the X toolkit. A widgets is user interface abstraction endowed with methods, private data, an X window, behavior, and semantics. These objects can be organized into a class hierarchy, and may be instantiated and combined to produce complex user interfaces.

G.1.2 Classes

This extension can use many different types of widgets. These are divided into a class hierarchy based on their resources and behavior. For the most part, what separates widget classes is not appearance but behavior. Actual appearance can vary greatly from instance to instance as well as from class to class, because the appearance of a widget instance is, for the most part, determined by the settings of its display attributes. All widgets share the same display attributes but these attributes may be set differently in each instance.

As an example of this, consider two widgets, one an instance of the #label class and the other an instance of the #command class. They would both appear on the screen
as small boxes with a bit of text within. It is possible that they could be the same size, shape and color, visually identical in every way.

The behavior of these widgets, though, are far from identical. The label is just that — a small text tag on the screen. It never exhibits any sort of behavior other than just “existing” at a particular location. The command button, though, is much more active. It responds to a sequence of user actions (usually entry of the mouse) by highlighting itself. Another sequence (usually a button press) will cause it to notify an application in some manner.

**Bindings** It is important to note this exact behavior is not “hardwired” into a widget. Instead, widgets contain a set of bindings that map lexical input (user events) to semantic actions. In X documentation, these bindings are often referred to as the translation table for a widget instance.

Different bindings can be specified for each widget instance. For example, you could create two different instances of a command button. The first instance might have the default translation table. This widget would highlight when the mouse entered it and notify when a mouse button is pressed.

The second instance could be a “fail safe” command widget. Bindings could be specified so that the user would have to hold down one mouse button to highlight the widget and then press a second mouse button to send back notification.

### G.1.3 Building with Widgets

**Widget Structures** The average X/Rose application will create complex structures such as menus and dialog boxes. These structures are built up as trees of widgets. That is to say, they are composed of large parent widgets having smaller children, having smaller children, having smaller children, . . . until the desired level of detail is reached. There is no reason why a child widget *must* be smaller than its parent, but the boundaries of the parent determine what portion of its children are visible. A widget is said to *clip* its children. This means that a portion of the child, which would extend beyond the border of the parent, will not be seen. This fact is used to advantage by certain widget classes. An instance of the `#viewport` widget class normally has a single, large child, only a small portion of which may be visible at any one time.

**Geometry Management** Many widgets have strategies for managing the screen placement of their children. These strategies are called Geometry Managers. The sole purpose of some widget classes is to provide this automatic layout management. The `#box` widget, for example, always arranges its children in horizontal rows, fitting as
many as possible in a row. The #form widget allows the user to select from several strategies: placement at a fixed distance from the top, bottom, or sides, or centering within the available space.

**Dealing with Children** Many operations on a widget also automatically affects its children. The positions of children are always maintained relative to their parent. As a rule of thumb, if an operation changes the visibility of a widget, the visibility of its children are likewise changed. An operations affecting the size of a parent, however, may not necessarily affect the size of its children.

It should be noted that not all widget classes are allowed children. Those that are allowed children are often referred to as *Composite* widgets. Instances of such classes as #command and #label are not allowed to have children.

### G.1.4 Class Resources

Each class can have associated with it some *resources*, publicly accessible instance variables. These resources are typically data values like strings and integers, or references to other objects like fonts, bitmaps, or graphics contexts.

As an example consider the label and command widget classes. Widgets of this class contain a text string. This string is an example of a class resource.

You should consult the widget summary for a complete description of resources available to each widget class.

### G.1.5 Class Operations

Each widget class may also define a set of operations generally applicable to instances of that class. Usually, these operations are intended to interpret or otherwise manipulate resources specific to the class. By convention, these commands are named Rx._Class._Operation. You should consult the command and widget summaries for more information about these.

As an example of class operations, consider the #gfx widget class. This class defines a set of drawing operations that use virtual world coordinates. The rx._gfx._lne and rx._gfx._pnt operations are applicable only to widgets of class #gfx. These operations take advantage of the special features of the #gfx class and could not function properly if applied to instances of other widget classes.
**G.1.6  Events**

**Mechanism** Each widget class can implement a set of semantic events that an application could be notified about. Many of these events that do not correspond directly to physical actions, rather they correspond to the semantics of the widget and thus, vary between classes. For example, a command button has a press event that can be triggered by a button press within the widget. This type of event would be meaningless for other types of widget.

Not all events must directly correspond to user actions. Every widget class, for example, has a destroy event. This type of event is triggered by the impending destruction of a widget instance.

The application may ask to be notified when an event is detected in a particular widget. When such an event is detected a RoseTalk string, provided by the application, will be executed. The system has a special operating mode, *Notify mode*, that allows this notification to take place. \[1\]

The system does not automatically monitor for every event in every widget. Rather, an application must inform system of those widgets and events that it is interested in. Furthermore, for every widget-event pair, it must provide the RoseTalk string to be executed upon detection of the event. The execution of this string is referred to as notification. It is a simple matter to start and stop notification for these widget-event pairs.

**Event Types** As mentioned earlier, each widget class has a set of events that it can be monitored for. Individual explanations of these can be found in the widget documentation. Because of possible name conflicts, when specifying an event type, you are required to supply both the name of the event and the name of the class that you are using.

When the application is notified, a supplied RoseTalk string is executed. The notification process for some events concatenates various parameter data to the string before execution. Thus the application can be passed event-specific data. The format and presence, if any, of parameters can be found in the widget documentation.

In addition to the events supported by each class, there is a set of events available to all widgets, regardless of class. These have been put into the system as the events of the pseudo-class *#xevent*. It is recommended that you use these events only when you have been unable to find satisfaction anywhere else. The reason for this is that these events are very primitive in nature, and thus, may bind the application too closely to the lexical details of the user interface.

---

1 Notify mode is the approximately equivalent to the AST mode of the UIS package
G.2 Resources

In general, an X resource is a set of values, or chunk of memory, that is kept within the server. These resources may be referenced by an integer identifier and are used by some functions. Resources are kept within the server for performance reasons. They are usually large or frequently used and would take too long to send across the network connection every time.

G.2.1 Graphics Contexts

An X graphics context is a block of information that controls how graphics operations will be executed. By nature, all graphics operations require a graphics context. This block of information contains such things as foreground and background color, tiling patterns and so forth. When a graphics context (GC) is created, the block of information is stored within the server, and an integer identifier is returned. The user may then reference this block of graphics information with this GC identifier. The use of GCs by X drastically reduces the amount of data that must be passed over the client-server connection.

Several GCs may be created and then interchanged to produce different output styles. In fact, this method is preferred over changing the values of a single GC. Swapping two different graphics contexts equivalent to swapping two integers and is considerably faster than sending a server request to change some value within a GC.

G.2.2 Fonts

An X font is a set of small bitmaps. These bitmaps (or glyphs) are used as clipmasks by the text operations. All glyphs for a font are stored in a file. Before a font can be used, this file must be read in from disk. The glyphs are stored in the server and the user is given an identifier to reference them.

G.2.3 Bitmaps and Pixmaps

An X Pixmap is an array of pixel values kept in off screen memory. On color systems a pixel value usually requires several bits, so a pixmap is divided into bit planes. A bitmap is just a single plane (one bit pixel value) pixmap. Pixmaps are kept by the server and referenced by a unique identifier. On some systems, off screen memory may be limited, so it is wise to use pixmaps sparingly.

Bitmaps, because of their simple structure, may be kept in a disk file and loaded when needed. This is a special case and is not possible for the more general pixmap. Normally, the data held by a pixmap must be generated at runtime.
G.2.4 Color

Color has not yet been made available from within X/Rose.
Appendix H

Using The Graphics Extension

This chapter is a introduction to the use of the X/Rose extension. This chapter should contain sufficient information to allow the reader to begin construction of an X/Rose application. Most X/Rose application are found to have several phases and a specific overall organization. Each of these phases will be briefly examined, and the overall organization X/Rose applications will be explained. Two sample applications are included at the end of this chapter to highlight and further demonstrate the ideas described herein.

H.1 Invoking Rose

To use the X/Rose extension, you should invoke Rose on a machine currently running an X server. It is also possible to have X output sent to a remote machine, but we will not discuss this here.  

From an xterm window, call the Rose executable by

```
rose {options}
```

The following command line switches will be of interest to you.

```
-g ....... Opens a connection with the X server and performs some initialization at the C level. If you do not use this option, none of the features in this manual will be available to you.
```

---

1 The DISPLAY environment variable controls the destination of output requests. By default, this specifies the X server on the local machine, but may be changed to the X server on any other network host.
This causes the file `start.ams` to be read in. This file loads all of the Rose definitions and routines used by the system. Once again, if you do not use this option, the features in this manual will be unavailable to you.

As of this writing, these are the only options available. It is expected that these might change slightly in the future to provide further services.

## H.2 Starting the System

Most X/Rose applications will go through the following phases. Each phase will be described in greater detail by the following sections.

- If needed, resources such as fonts, graphics contexts, and bitmaps are loaded or created.
- Then widgets will be instantiated and customized.
- The user will specify, for each widget, how he wants events to be handled.
- Widgets will be placed on the screen and the application will enter a state where it simply reacts to asynchronous user input.

### Identifiers

In this system, objects such as widgets, graphics contexts, fonts and other resources are assigned unique numeric identifiers. You will be required to use these IDs whenever referencing an object.

Unfortunately, these ID’s (as well as the objects they reference) are not persistent. They are valid only during the current Rose session. This means that at the beginning of any Rose application, you will have to rebuild or recreate these resources to get a new set of valid identifiers.

## H.2.1 Allocating Resources

Resources such as fonts, bitmaps, and graphics contexts are allocated using a variety of functions. These functions generally return a single, unique integer identifier for each resource. It is with these identifiers that resources are specified as arguments to other functions and procedures.

A default graphics context is provided for you, but for all but the simplest of applications, additional graphics contexts are usually needed. A graphics context is created using the `rx_create_gc` function, which accepts a wide variety of parameters, specified
as a list of name/value pairs. This allows you to specify as many or as few options as are needed.

Fonts must be loaded into the server before they may be used. This is done with the \texttt{rx\_load\_font} command, which takes a font name and returns an identifier. In addition there are some system functions that allow you to list the available fonts in the system and to direct the server to custom designed fonts.

Bitmaps and pixmaps may be created with currently undefined functions. See the command summary for details. In addition, bitmap data may be read in from disk using the \texttt{rx\_load\_bitmap} function. This function takes a path and filename and returns a bitmap identifier.

\section*{H.2.2 Creating Widgets}
You create widgets by using the \texttt{rx\_create\_widget} function, which takes the identifier of a parent widget, a class name, a list of resource name/value pairs, (See Setting Resources) and returns the identifier for the new widget. You can use this command and the resulting widgets to build a widget structure to suit your needs.

\textbf{Organizing the Widgets} The widget instances of your application will form one or more trees. At the root of such a tree will be a “top level” widget. This widget acts as a frame or bounding box for your other widgets and is placed directly on the root window of your display. The window manager will then treat each tree of widgets as a separate window for things like moving, resizing and iconifying. There is no limit on the number of widget trees you may create, but most programs use only one. They create a large top level widget and then subdivide it into several sections. A top level widget is created by specifying a null parent to \texttt{rx\_create\_widget}.

The main reason for only using one is that, for the most part, the user, \textit{not} the application, controls the size and placement of top level widgets. Forcing a user to keep track of several windows from each application can become inconvenient and confusing.

Note that, at this point, the widgets have not yet appeared on the screen. Only the internal representations of widgets have been created. As you will notice, widgets do not appear on the screen until after they have been \textit{realized}.

\section*{H.2.3 Setting Resources}
At creation time, a widget’s attributes are set from an internal resource database and from values passed to \texttt{rx\_create\_widget}. The database contains system defaults as well as any values the user may have specified through defaults files and command line
options. At some point you may want to modify some of these values for a particular widget.

The usual mechanism for setting these resources is the rx_set_values command. This command is flexible enough to handle most resources. It is used by providing the identifier for the desired widget as well as a list of names and values for those resources you wish to change.

Some special purpose functions are also available. These are intended for those cases where it is not convenient to use the format required by rx_set_values. An example of such a situation is the rx_setBindings command. This command accepts a list of user inputs and actions, as well as a widget ID. For the given widget, the specified actions are then bound to the given input sequences.

Finally, there is a mechanism for setting system-wide values. This is done in a manner similar to that used by rx_set_values. The rx_set_defaults command takes a list of default names and corresponding values.

### H.2.4 Selecting Notification

If you would like to be notified of events within a certain widget, you must follow a two step procedure. First, you must let the system know what events you are interested in. This is done with the rx_create_notify command. You supply the ID of the widget you want to monitor, a description of the event you are interested in, and the action to take (a RoseTalk string) when such an event occurs. The command returns a notify ID representative of this particular combination of widget, event, and action.

The second step is to tell the system to begin monitoring for a specific widget and event combination. This is done with the rx_start_notify command. This command takes a list if notify IDs and activates them. From this point on, whenever an activated widget/event combination is detected, the action specified for this pair will be placed in a command queue. Note however, that these actions are executed only when the system is in Notify mode.

It is likewise possible to **Deactivate** widget/event combinations. This is done with rx_stop_notify. Given a list of notify ID’s, this command tells the system to stop monitoring for these conditions.

**Maintenance** At some time, you may wish to change the conditions represented by a notify ID. The following maintenance commands have been made available for this purpose.

- rx_change_notify_wid will change the widget to be monitored.
• \texttt{rx\_change\_notify\_event} will change the event that is monitored for.
• \texttt{rx\_change\_notify\_action} will change the RoseTalk string that is executed upon notification.

H.2.5 Realizing Widgets

After you have created all the widgets that you will need, you should call \texttt{rx\_realize\_tree} on all top level widgets. This recursively descends through a widget tree and places each widget on the screen.

You should only call this once in an application because once a widget is realized, any new children will also be automatically realized. In spite of this, you are encouraged to create as many of your widgets as possible before realizing them. During the creation procedure, widget attributes may be changed several times before reaching a stable value. Repeating changes on the screen is much less efficient than just changing its internal values.

H.2.6 Main Loop

At this point, the typical application will enter Notify mode and simply react to asynchronous user input.

H.3 A Simple Example

/* File: small\_test.ams
/*
/* A starter program ...
/*

Create a widget tree. The structure will be a command button that, when pressed, will cause rose to drop out of ast mode.

/* Create a button on the root window
   BUTTON := rx\_create\_widget {0, \#command,
+ "label" has\_value "Press Me"}

Tell the system that we want to be notified when the command button is activated. In particular, we want the system to inform us by executing the command \texttt{ast \{\}}.
/* Set up notification
   N1 := rx_create_notify {BUTTON, #command, #press, "ast {} /*"}

Everything has been set up, so we now send the widget to the screen, tell the system
to begin sending us notification, and simply react to any user actions.

/* Flush everything to the screen and react to user input

   rx_realize_tree {ROOT}
   rx_start_notify {N1}
   ast {}

H.4 A More Complex Example

/* File: large_test.ams
/*
/* Tests many X/Rose features ...
/*
object_set {#integer, #integer}

Allocate any necessary resources. Load some bitmaps and fonts then create several
graphics contexts.

/* Create a scaly reptilian bitmap
   BITS := rx_load_bitmap("/usr/include/X11/bitmaps/scales")
   EYE := rx_load_bitmap("/usr/include/X11/bitmaps/target")

/* Create several graphics contexts
$ procedure gc {}

   GER := rx_load_font{"ger-s35"}
   F1 := rx_load_font{"6x10"}
   F2 := rx_load_font{"6x12"}
   F3 := rx_load_font{"6x13"}
   F4 := rx_load_font{"8x13"}
   F5 := rx_load_font{"9x15"}

   G := rx_create_gc {
      +#GCFont has_gc_value F1
   }
   G2 := rx_create_gc {

Create a widget tree. The structure will be a box with some buttons a graphics window and a scaly background. The graphics window will contain a graphics sub-window with a funky border.

/* Save gc ids for later use
   insert {#integer, #normal, G}
   insert {#integer, #fat, G2}
   insert {#integer, #dash, G3}
   insert {#integer, #fill, G4}
   insert {#integer, #highlight, G5}
$ 

gc {}
W5 := rx_create_widget {ROOT, #scroll,
+   ("orientation" has_value 0) cat
+   ("height" has_value 20) cat
+   ("width" has_value 220) cat
+   ("thickness" has_value 30)}

WIN := rx_create_widget {ROOT, #gfx,
+   ("height" has_value 300) cat
+   ("width" has_value 200)}

TIN := rx_create_widget {WIN, #gfx,
+   ("height" has_value 50) cat
+   ("width" has_value 50) cat
+   ("borderWidth" has_value 5) cat
+   ("borderPixmap" has_value EYE) cat
+   ("x" has_value 20) cat
+   ("y" has_value 20)}

/* Save widget ids for later use
   insert {#integer, #W2, W2}
   insert {#integer, #WIN, WIN}

/* Change the behavior of the "Exit Ast" button
   rx_set_bindings{W1,
+   ("<Btn1Down>" has_binding "set()") cat
+   ("<Btn3Down>" has_binding "notify() unset()") cat
+   ("<EnterWindow>" has_binding "highlight()") cat
+   ("<LeaveWindow>" has_binding "unset(NoRedisplay) unhighlight()")}

Select those events that we want notification on and specify the procedures to handle them. In particular, we specify procedures for each button press, for the motion of one slidebar, and to refresh the contents of the graphics window.

/* Set up notification
   N1 := rx_create_notify {W1, #command, #press, "ast {} /*"}
   N2 := rx_create_notify {W2, #command, #press, "msg {"hey"} /*"}
   N3 := rx_create_notify {W3, #command, #press, "my_fun {}"
   N4 := rx_create_notify {W4, #scroll, #thumb, "scroll_handler {}"
   N5 := rx_create_notify {WIN, #xevent, #Expose, "expose_handler {} /*"}

/* Notification handlers
$ procedure my_fun {PARAMS}
msg {"Inside my_fun"}
msg {"Notify ID: " one'\merge (one'string PARAMS[0])}
msg {"Widget ID: " one'\merge (one'string PARAMS[1])}

$ procedure scroll_handler {IDS, POS}
   rx_set_values {#W2%integer, ("label" has_value one'string POS)}
$

$ procedure expose_handler {}
   msg {"Expose-handler"}

   rx_set_defaults { 
      + (#dflt_wid has_value #WIN%integer) cat
      + (#dflt_gc has_value #normal%integer)
      rx_circle {150, 100, 25}
      rx_circle {50, 100, 25}
      rx_line {100, 100, 100, 200}
      rx_arc {50, 200, 100, 40, 180*64, 180*64}
      rx_draw_string {10,180,"Tiny"}
   
   rx_set_defaults { (#dflt_gc has_value #fat%integer) }
   rx_line {25, 20, 175, 20}
   rx_draw_string {10,150,"UberRose"}

   rx_set_defaults { (#dflt_gc has_value #dash%integer) }
   rx_line {0, 250, 150, 250}
   rx_draw_string {10,200,"Small"}

   rx_set_defaults { (#dflt_gc has_value #highlight%integer) }
   rx_line {0, 100, 100, 100}
   rx_draw_string {10,220,"Bigger"}

   msg {"Done"}
$

Everything has been set up. Initialize the widgets and send them to the screen, start up the notification, and react to any user actions.

/* Flush everything to the screen and react to user input */

   rx_realize_tree {ROOT}
H.5 Limitations

When running Rose in notify mode, you should have no trouble with any of the features described in this manual. When not in notify mode, the screen display may not behave properly. Widgets may not react as they are supposed to, and the results of graphics may not appear on the screen immediately. This is due to the queuing systems described earlier.

What might happen is that your graphics operations may be sit in the output queue, waiting for block transmission to the server. As such, the server would not yet know of your graphics requests. Also, a widget may not be exhibiting proper behavior because it has not yet been informed of some user input. This user input may have been received from the server, but it could still be sitting in the input queue. These input and output queues are ignored as Rose waits for your input.

One possible remedy for this behavior is to call \texttt{rx\_flush\_req} when output operations are complete. This has the effect of flushing the output queue, sending all pending graphics operations over to the server.

Unfortunately, this still does not help the widget behavior. For the most part, widgets that respond to some user actions (like a \texttt{#command} widget) are not usable outside of Notify mode. These widgets will not respond interactively and any notification that they will return will not be processed.

For prototyping purposes, though, the \texttt{rx\_sync} command has been provided. This command brings all widgets up to date by clearing both input and output queues. By looping and calling this command at the start of each iteration, it is possible to simulate the interactive nature of these widgets. This is not recommended though, because of the performance penalty inherent to the \texttt{rx\_sync} command. In addition, the notify facilities of these widgets would still be unavailable.

H.6 Suggestions

- While in notify mode, all operations and widgets should function fine. You do not need to use the \texttt{rx\_sync} or \texttt{rx\_flush\_req} commands.

- Create all of your widgets before calling the \texttt{rx\_realize\_tree} operation.
• Use these widgets to their full potential!
  You have been provided with some powerful and flexible tools. Modify a wid-
geet’s bindings to customize its behavior. Only as a last resort should you at-
tempt to customize its behavior by requesting notification for the \textit{Xevent} class events.
Appendix I

Command Summary

The following is a listing of all commands currently available within this system. The calling syntax is presented, along with descriptions of each parameter. The usage of each command is explained, often with example call, and possible trouble spots are pointed out.

These commands have been subdivided into the following groups. In addition to these, class specific commands can be found in the widget summary.

- Widget Manipulation
- Notification
- Resources and Utilities
- Graphics
- Other/Internal
I.1 Widget Manipulation

I.1.1 Rx_Create_Widget

Function rx_create_widget {parent, class, vals}
  class .... Class name of new widget
  parent .. ID of parent widget
  vals ..... List of resource names and values
Returns .. Integer, new widget identifier

Use: This command takes the given class and parent, and attempts to create a new widget instance. The resource/value pairs given by vals are used to configure the new widget. See rx_set_values for a complete description of format for the vals argument. For a complete list of available classes and valid resource names, refer to the widget documentation.

Example:

ROOT := rx_create_widget {0, #box,0}
W1 := rx_create_widget {ROOT, #command,
  ("label" has_value "Press Me")}

W2 := rx_create_widget {ROOT, #gfx,
+   ("height" has_value 300) cat
+   ("width" has_value 200)}

This example creates a top-level box widget with two children. The first child is a command button containing the label “Press Me” and the second child is a 200 pixel wide, 300 pixel high graphics box.

I.1.2 Rx_Destroy_Widget

Procedure rx_destroy_widget {wids}
  wids ..... List of widget IDs

Use: This command permanently removes the widgets given by wids and all of their children from the screen, and then destroys them. Any further reference to these widgets will result in an error. Use this command only when widgets will never again be needed. To temporarily remove a window from the screen, use rx_hide

Notes: This command does not do anything about existing notify records. In the future, this should probably track down and delete any hanging references.
I.1.3 Rx_Realize_Tree

Procedure rx_realize_tree \{wids\}
   wids ..... List of widget IDs (usually top level widgets)

Use: Initialize a widget tree. When a top level widget (widget with a null parent) is created, it is not automatically placed on the screen. Instead, this widget and all of its children remain unrealized. These widgets are not placed on the screen until they are realized via this procedure. The reason for this is that while creating a widget tree, parents often move and resize their children. For efficiency, widgets should not be placed on the screen until final size and placement has been decided upon. This is normally only called once for all top level widgets.

I.1.4 Rx_Hide

Procedure rx_hide \{wids\}
   wids ..... List of widget IDs

Hide a widget. This command temporarily removes the widgets given by \texttt{wids}, and all their children, from the screen. They can be put back on the screen with the \texttt{rx_unhide} command.

Notes: Widgets that are already hidden will be unaffected by this command.

I.1.5 Rx_Unhide

Procedure rx_unhide \{wids\}
   wids ..... List of widget IDs

Unhide a widget. This command makes previously hidden widgets, given by \texttt{wids}, visible again. The widgets will reappear in the same place as they exactly the same

Notes: Widgets that are already visible will be unaffected by this command.
I.1.6 Rx_Popup

Procedure rx_popup {spec}

   spec ...... widget/grab type pairs

Placed a popup class widget on the screen. The widgets to be made visible are given
by spec. They can be removed from the screen with the rx_popupdown command. The
spec widget/grab pairs are built with the has_grab function which has the following
format:

   ( widget-id-list has_grab grab-type )

The grab types indicate how events are to be dispatched to the widgets. The grab
types may be one of the following:

    #GrabNonExclusive ....
    #GrabExclusive ..........  
    #GrabSpringLoad .......

If any widgets are popped up with #GrabSpringLoad events will be redirected to
the most recently popped up one. Otherwise, if any widgets are popped up with
#GrabExclusive, then only events that occur within those widget trees are delivered.
If nothing is popped up, or if everything is popped up with #GrabNonExclusive, then
events are delivered normally.

Notes: The Sun X implementation contains bugs and currently ignores the grab
type. Widgets must be of type popup to be affected by this command.

I.1.7 Rx_Popup_Relative

Procedure rx_popup_relative {spec, wid, x, y}

   spec ...... Widget/grab type pairs
   wid ...... Widget to use as relative origin
   x ......... X distance in pixels
   y ......... Y distance in pixels

This command is identical to rx_popup, but places a popup class widget on the screen
at the offset (x,y) from the upper left corner of wid. The widgets to be made visible
are given by spec

Notes: Widgets must be of type popup to be affected by this command.
I.1.8  Rx_Popdown

Procedure rx_unhide \{wids\}

\textit{wids} \ldots \ldots \text{List of widget IDs}

Removes a popup from the screen. This command makes previously popped up widgets, given by \textit{wids}, invisible again.

I.1.9  Rx_Set_Defaults

Procedure \texttt{rx_set_defaults} \{\texttt{vals}\}

\textit{vals} \ldots \ldots \text{List of default names and values}

Use: This command is the generic mechanism for setting various system defaults. The widget instance data, known as resources, and the associated values are specified by the \texttt{vals} list. A change to a particular default is specified by the name of the default and new value. The value can be either an integer real, string, or a named constant. The default name/value pairs are constructed with the \texttt{has_value} function which has the following format:

\[
( \text{default-name has_value value} )
\]

These pairs may then be strung together with the \texttt{cat} command.

Example:

\texttt{rx_set_defaults \{(\#default_wid has_value W1) cat (\#default_gc has_value GC2)\}}

This example sets the default widget to the id contained in \texttt{W1} and the default graphics context to the id contained in \texttt{GC2}.

Valid parameter names are as follows:

- \texttt{\#default_wid}  Widget to be used by most graphics operations. This expects a widget id. No default widget is automatically provided, so this must be set before any graphics operations are used.

- \texttt{\#dflt_wid}  Same

- \texttt{\#default_gc}  Graphics context to be used by most graphics operations. This expects a valid gc id. A basic gc is automatically provided by the system, so this need not be set unless the user wishes to use a gc with different attributes. Once it has been set to some custom gc, the original, basic gc may be restored by setting this to zero.
#dflt gc  Same

#default_drawmode  Drawing mode to be used by the rx_connected_lines procedure. This specifies how the coordinate list is to be interpreted. There are two options, take each coordinate as an offset from the origin or as an offset from the previous point.

  #:draw_absolute  ........ Offset from the origin
  #:draw_relative  ........ Offset from the last point

#default_ntfy_fmat The format for the string placed in the Rose command buffer. This format string must be valid for use in a C printf function call. The format should accept two integers (Notify ID twice) and a string (possible return parameters). The format string normally takes the form of an execute{} command with a call to some mapping. This mapping looks up the action with the first notify ID and merges it with the second notify ID, and the parameter string.
I.1.10 Rx_Set_Values

Procedure rx_set_values {wid, vals}
   wid ...... Single widget ID
   vals ...... List of resource identifiers and values

Use: This command is the generic mechanism for setting the values of various widget instance data. The widget instance is specified by wid. The widget instance data, known as resources, and the associated values are specified by the vals list. A change to a particular resource is specified by a resource name and new value. A resource name is simply a string. The value can be either an integer real, string, or a named constant. The resource name/value pairs are constructed with the has_value function which has the following format:

   ( resource-name has_value value )

These pairs may then be strung together with the cat command.

Example:

rx_set_values {W1, ("label" has_value "Exit AST") cat
   ("height" has_value 100) cat
   ("width" has_value 200)}

This example sets the height of W1 to 100 pixels, its width to 200 pixels, and its label to “Exit AST”. Typical resources include things like size, position, border background pattern, and labels. You should refer to Chapter 6 for complete lists of valid resource names and their formats.

Notes: This facility is fairly forgiving. If a resource does not apply to a particular widget class, references to it will be ignored. The operation invokes some internal integrity enforcement operations, so to optimize performance it is best to batch changes into one call.
I.1.11 Rx_Get_Values

Function rx_get_values {wid, vals}

wid ...... Single widget ID
vals ...... List of resource identifiers and types

Returns .. A list of strings integers and reals (the requested values) enclosed into an OR structure.

Use: As the name implies, this is the opposite of the RxSetValues command. This command provides a generic mechanism for retrieving selected data values in widget instance data. The widget instance is specified by wid. The vals list is used to request the resources that should be returned. A request is composed of a resource name and a type name. The resource names are the same ones used in the RxSetValues command, and the type names are one of #integer, #string, or #real.

The resource/type pairs are constructed with the has_type function which has the following format:

( resource-names has_type type-name )

These pairs may then be strung together with the cat command.

Example:

VALS := rx_get_values {W1, ("x" "y" has_type #integer) cat
"label" has_type #string)}

This example retrieves the (x,y) coordinates and the label of widget W1.

Notes: This facility is not as flexible as RxSetValues. The user must provide the correct type for each argument requested. This may be a bit bothersome, but the user would need this type information in any event, in order to properly process the returned values. A more serious restriction involves the use of named constants. When setting a value, the user is allowed to specify a named constant, but when retrieving, integer values can not be automatically translated back to named constants.
I.1.12 Rx_Set_Bindings

Procedure rx_set_bindings {wid, bindings}

\[ wid \ldots \text{Single widget identifier} \]
\[ bindings \quad \text{List of event and action names} \]

Use: This command used to set the mapping of user events to a widget’s actions. The widget instance is specified by \texttt{wid}, and \texttt{bindings} is a list of string pairs representing events and corresponding widget actions. These event/action pairs are constructed with the \texttt{has_binding} function which has the following format:

\[ ( \text{event} \text{ has_binding } \text{action} ) \]

These pairs may then be strung together with the \texttt{cat} command.

Example:

\begin{verbatim}
rx_set_bindings{W1, 
+   ("<Btn1Down>" has_binding "set()") cat
+   ("<Btn3Down>" has_binding "notify() unset()") cat
+   ("<EnterWindow>" has_binding "highlight()") cat
+   ("<LeaveWindow>" has_binding "unset(NoRedisplay) unhighlight()")}
\end{verbatim}

This example sets the bindings of the \texttt{W1} command widget so that the user must enter the widget and then press first the leftmost mouse button and then the rightmost mouse button to cause the command widget to “activate”.

You should refer to Chapter 6 for complete lists of actions and the event names.

I.1.13 Rx_Add_Bindings

Procedure rx_add_bindings {wid, bindings}

\[ wid \ldots \text{Single widget identifier} \]
\[ bindings \quad \text{List of event and action names} \]

Use: This command behaves identically to \texttt{rx_set_bindings}, but instead of setting a widgets bindings to only those specified in \texttt{bindings}, \texttt{rx_add_bindings} adds to the existing bindings, preferring the new bindings if there is any conflict.
I.1.14 Rx_Set_Sensitive

**Procedure rx_set_sensitive {specs}**

*s specs .... widget/sensitivity pairs

**Use:** This command is used to switch widgets on or off. When a widget’s sensitivity is set to true, events are dispatched and handled in the normal manner. When the sensitivity is set to false however, mouse and keyboard events are not sent to the widget. That widget, and all of its children, becomes *insensitive* to user actions. Any widget can be made sensitive or insensitive to user events. When insensitive, some widget classes change appearance, perhaps appearing stippled or blanked.

The widget/sensitivity pairs are constructed with the *has_sensitive* function which has the following format:

\[
( \text{resource-name} \ \text{has_sensitive} \ \text{value} )
\]

These pairs may then be strung together with the *cat* command.

**Example:**

OFF := (#w1 #w2 #w3) %widget_ids
ON  := (#w4 #w5) %widget_ids
rx_set_sensitive {((OFF has_sensitive #false) cat
                  (ON has_sensitive #true))}

This example sets widgets #w1, #w2, and #w3 so that they are insensitive, and sets widgets #w4, and #w5 so that they are sensitive,
I.2 Notification

I.2.1 Rx_Create_Notify

Function \texttt{rx_create_notify} \{\textit{wid}, \textit{class}, \textit{event}, \textit{action}\}
\begin{itemize}
  \item \textit{wid} ........ Single widget ID
  \item \textit{class} .... Class name
  \item \textit{event} .... Event name
  \item \textit{action} .. Valid RoseTalk string to be executed
\end{itemize}
Returns .. Integer, new identifier for this set of conditions

Use: Specify event conditions to be notified on. Creates an entries in the Notify data set specifying events to monitor for and appropriate actions to take. Notification will not start until \texttt{rx_start_notify} is called. Notification for some events will provide you with parameters. You should plan for this when specifying your action strings. Parameters, if any, will be concatenated to your action string along with a closing brace. Action strings for such events should consist of a procedure or function name, an opening brace, and any parameters that you provide. When executed, the system will add the remaining parameters and a closing brace.

Example:

\texttt{N1 := rx_create_notify \{W1, \#command, \#press, "msg \{"Hi\}" /"\} \}}
\texttt{N2 := rx_create_notify \{W2, \#xevent, \#Motion, "move\{1 2 3, " \} \}}

In these examples \texttt{W1} is an instance of \texttt{\#command} class. The first example requests notification on the \texttt{\#command} class press event. When such an event is received the message “Hi” will be printed. Note the comment symbol at the end of the action string. This hides the closing brace and any parameters that the system might add.

The second example requests notification on mouse movement. When such events are received the procedure “move” will be called with the parameter “1 2 3” and a list of position data to be supplied by the system.

I.2.2 Rx_Destroy_Notify

Procedure \texttt{rx_destroy_notify} \{\textit{nids}\}
\begin{itemize}
  \item \textit{nids} ...... List of notify IDs.
\end{itemize}

Use: Removes the given object from the Notify data set. This command destroys the specified notify records. Any active notify records will be deactivated before they are destroyed. To temporarily suspend notification, use \texttt{rx_stop_notify}
I.2.3  **Rx_Start_Notify**

Procedure `rx_start_notify {nids}`

nids .... List of notify IDs.

**Use:** Start notification for the widget and event pairs referenced by `nids`. Also known as *Activating* the notify IDs in `nids`.

I.2.4  **Rx_Stop_Notify**

Procedure `rx_stop_notify {nids}`

nids .... List of notify IDs.

**Use:** Halt notification for the widget and event pairs referenced by `nids`. Also known as *Deactivating* the notify IDs in `nids`.

I.2.5  **Rx_Change_Notify_Action**

Procedure `rx_change_notify_proc {nids, actions}`

nids .... List of notify IDs.

procs .... List of RoseTalk strings.

**Use:** Changes notification conditions. For each notify record named in the `nids` list, the RoseTalk executable is changed to the corresponding string in the `actions` list. Both active and inactive notify records may be modified with this procedure.

I.2.6  **Rx_Change_Notify_Wid**

Procedure `rx_change_notify_wid {nids, wids}`

nids .... List of notify IDs

wids .... List of widget IDs

**Use:** Changes notification conditions. For each notify record named in the `nids` list, the widget to be monitored is is changed to that given by the `wids` list. Active notify records may not be modified with this procedure.
I.2.7 Rx_Change_Notify_Event

Procedure rx_change_notify_event {nids, classes, events}
   nids ..... List of notify IDs
   classes . List of class names
   events .. List of event names

Use: Changes notification conditions. For each notify record named in the nids list, the triggering condition is changed to the corresponding event in the classes and events lists. See the widget class documentation for a description of valid events. Active notify records may not be modified with this procedure.
I.3 Resources and Utilities

I.3.1 Rx_Create_GC

Function \texttt{rx_create_gc \{vals\}}

\begin{itemize}
  \item \texttt{vals} \ldots \ldots \text{List of values for the new graphics context}
  \item \text{Returns} \ldots \text{Integer, new graphics context identifier}
\end{itemize}

Use: Create a graphics context for future use. A graphics context is used to control the results of the various drawing primitives. This command takes a series of parameter name/value pairs and returns an integer identifier for the new graphics context. The parameter name/value pairs are constructed with the \texttt{has\_gc\_value} function which has the following format:

\begin{itemize}
  \item \texttt{( parameter-name has\_gc\_value value )}
\end{itemize}

These pairs may then be strung together with the \texttt{cat} command.

Example:

\[
\text{GC := rx_create_gc \{ (#GLineWidth has\_gc\_value 5) cat + (#GLineStyle has\_gc\_value #LineOnOffDash)\}}
\]

This example creates a graphics context for drawing dashed lines that are 5 pixels thick.

Notes: Use the \texttt{rx_set_defaults} command with the parameter name \texttt{#default\_gc} to specify a graphics context for subsequent drawing operations.

The following is a listing of valid parameters along with a description and some possible values.

\begin{itemize}
  \item \texttt{#GFunction} Controls how the source (src) pixel values combine with the destination (dst) pixel values to produce a final pixel value. This parameter can take any of the following named values.
    \begin{itemize}
      \item \texttt{#GXclear} \ldots \ldots \ldots Clear pixel
      \item \texttt{#GXand} \ldots \ldots \ldots src \texttt{and} dst
      \item \texttt{#GXandReverse} \ldots \ldots src \texttt{and not} dst
      \item \texttt{#GXcopy} \ldots \ldots \ldots src
      \item \texttt{#GXandInverted} \ldots \ldots (not src) \texttt{and} dst
      \item \texttt{#GXnoop} \ldots \ldots \ldots dst
      \item \texttt{#GXxor} \ldots \ldots \ldots src \texttt{xor} dst
      \item \texttt{#GXor} \ldots \ldots \ldots src \texttt{or} dst
    \end{itemize}
\end{itemize}
#GXnor . . . . . . . . . . . . . (not src) and not dst
#GXequiv . . . . . . . . . . . . . (not src) xor dst
#GXinvert . . . . . . . . . . . . . not dst
#GXorReverse . . . . . . . . . src or not dst
#GXcopyInverted . . . . . not src
#GXorInverted . . . . . . . . . (not src) or dst
#GXnand . . . . . . . . . . . . . . (not src) or not dst
#GXset . . . . . . . . . . . . . Set pixel

#GCPlaneMask Specifies what bit planes in a pixel are affected by an operation. This parameter takes an integer value, where each bit of the integer corresponds to a possible plane in the source and destination pixels.

#GCClipMask Specifies a bitmap to clip the action of a graphics operation. This parameter takes a valid bitmap identifier. (cf. rx_create_pixmap or rx_load_bitmap)

#GCClipXOrigin Specifies where to use the clipping bitmap.

#GCClipYOrigin Specifies where to use the clipping bitmap.

#GCForeground Specifies foreground color. This parameter takes a valid color identifier. (cf. rx_create_pixel)

#GCBackground Specifies background color. This parameter takes a valid color identifier. (cf. rx_create_pixel)

#GCLineWidth Specifies line width, measured in pixels. There is one special case to this parameter. A line width of zero results in a one pixel wide line that uses device-dependant algorithms for speed. As such, it is usually much faster than using a line width of one. The device-dependant algorithms, however, may produce slight inaccuracies. For quick one pixel lines, one should use a line width of zero. For accurate one pixel lines, one should use a line width of one.

#GCLineStyle Specifies if and how lines should be dashed. See #GCDashList to set the dash size. This parameter takes one of the following name values.

#LineSolid . . . . . . . . . . . . . Even and odd segments are filled the same

#LineDoubleDash . . . . . . . . . Even and odd segments are filled differently. See #GCFillStyle for details
#LineOnOffDash ....... Only even segments are drawn

#GCCapStyle Specifies how lines are to terminate. This parameter takes one of the following name values.

#CapButt ............... Square end.
#CapNotLast ............ Same as above, but single pixel width lines are drawn with the final pixel omitted
#CapRound ............. Round end with diameter equal to the line width
#CapProjecting ......... Square end, projecting beyond the endpoint for a distance of half the line width

#GCJoinStyle Specifies how to draw the corners of multiple connected lines. Used by the rx_connected_lines command. This parameter takes one of the following name values.

#JoinMiter ............. Angular corner
#JoinRound ............ Arc with diameter equal to the line width
#JoinBevel ............ Angular but beveled down

#GCFillStyle Specifies what source graphics data will be drawn with. This parameter takes one of the following name values.

#FillSolid ............... Drawn with foreground color
#FillTiled ............... Fill with the #GCTile pixmap
#FillStippled ........... Drawn with the foreground color masked by the #GCStipple bitmap
#FillOpaqueStippled ... Fill with the #GCStipple bitmap, where the set and unset bits correspond to foreground and background colors, respectively

#GCFillRule Specifies which pixels are to be considered for fill requests. May take one of the following name values.

#EvenOddRule ............
#WindingRule ............

#GCTile Specifies a pixmap to be used for fill requests. This parameter takes a valid pixmap/bitmap identifier. (cf. rx_create_pixmap or rx_load_bitmap)

#GCStipple Specifies a bitmap to be used as a mask by some operations.
This parameter takes a valid bitmap identifier. (cf. rx_create_pixmap or rx_load_bitmap)

#GCTileStipXOrigin  Specifies where to use the Stipple and Tile bitmaps.

#GCTileStipYOrigin  Specifies where to use the Stipple and Tile bitmaps.

#GCFont  Specifies the font to use in text operations. This parameter takes a valid font identifier. (cf. rx_load_font)

#GCSubwindowMode  Specifies whether graphics operations on a widget are clipped by its children. This parameter takes one of the following name values.

  #ClipByChildren  .......  This is the normal state. The results of an operation may be obscured by any children

  #IncludeInferiors  .......  This ignores any children and draws over the top of them if need be

#GCDashOffset  Specifies how many pixels into a line a dash pattern should begin. This parameter takes an integer value.

#GCDashList  Specifies the length of a single dash or space in pixels. This parameter takes an integer value.

#GCArcMode  Specifies how arcs are to be filled. Takes one of the following name values.

  #ArcPieSlice  ............  The two lines joining endpoint to center are used as boundaries

  #ArcChord  ............  The line joining both endpoints is used as a boundary
### I.3.2 Rx_Copy_GC

Procedure `rx_copy_gc {src, dst, fields}`

- `src` ....... Single gc identifier, source
- `dst` ....... Single gc identifier, destination
- `fields` .. List of field names to copy

**Use:** Copy selected values of a graphics context. This command takes the fields named by `fields` and copies them from `src` to `dst`. See `Rx_Create_GC` for a listing of valid field names.

**Example:**

```
rx_copy_gc {G1, G2, #GCForeground #GCBbackground}
```

This example copies the foreground and background components of G1 to G2.

### I.3.3 Rx_Set_GC

Procedure `rx_set_gc {gc, vals}`

- `gc` ......... Single graphics context identifier
- `vals` ...... List of values for the new graphics context

**Use:** Set the values of a graphics context. This command takes a series of parameter name/value pairs. The parameter name/value pairs are constructed as for `Rx_Create_GC`. See the preceeding section for detail explanations.

**Example:**

```
rx_set_gc {G2, (#GCLineStyle has_gc_value #LineSolid) cat + (#GCCapStyle has_gc_value #CapRound)}
```

This example changes the line and cap styles of the graphics context given by G2.
I.3.4 Rx_Get_GC

Function rx_get_gc {gc, fields}

gc ....... Single gc identifier
fields .. List of field names to return
Returns .. List of integer values

Use: Get selected values from a graphics context. This command takes the fields named by fields and retrieves their values. The resulting integers are compiled into a list and returned. See Rx_Create_GC for a listing of valid field names.

Example:

WIDTH := rx_get_gc {G1, #GCLineWidth}

This example returns the line width setting of G1

Notes: This function will not return the named costants used for many fields, but rather the integer equivalents. If you wish to compare the result to a named constant, you must first convert the named constant to an integer.

I.3.5 Rx_Load_Font

Function rx_load_font {fonts}

fonts .... List strings representing font names
Returns .. List of integer font identifiers

Use: Loads a font file and returns an integer identifier for it. This font may then be used by setting the #GFont parameter of a graphics context to the returned identifier.

Example:

FONT_ID := rx_load_font{"9x15"}
GC := rx_create_gc {(#GFont has_value FONT_ID)}
rx_set_defaults {#default_gc has_value GC}

This example loads the 9x15 font, creates an new graphics context with containing the 9x15 font, and then makes the newly created graphics context the current one. Subsequent text operations will use the 9x15 font.
I.3.6 Rx_Destroy_Font

Procedure rx_destroy_font {fids}
    fids ..... List of font identifiers

Use:  Deallocates the fonts given fids. This should only be used when a font will
      not be needed again. Any further reference to the identifiers in fids will cause an
      error.

I.3.7 Rx_List_Font

Function rx_list_font {patterns}
    patterns List of pattern strings
    Returns . . List of strings, any matching font names

Use:  This command takes list of pattern strings and searches the available fonts
      for all matches. The list of matching font names are then returned. These names
      may be used with the rx_load_font to read in a specific font. The pattern strings may
      contain “*” and “?” which have the usual wildcard meanings.

Example:

FONT_NAMES := rx_list_font {"s*" "f*"}

This example returns the names of all available fonts which begin with either s or f.
FONT_NAMES will contain such names as serif10, serif12, fg-16, fg-18, and fg-20.

I.3.8 Rx_Get_Font_Path

Function rx_get_font_path {}
    Returns . . List of strings, directories in the font search path

Use:  Returns the search path used the the X server to find font files. The average
      user will never be concerned about this.
I.3.9 Rx_Set_Font_Path

Procedure rx_set_font_path {paths}

paths .... List of directories

Use: Sets the search path used by the X server to find font files. The only time this feature would be needed is if an application needed a custom designed font that was not kept with the usual font files. The average user will never be concerned about this.

Example:

rx_set_font_path {"/usr/lib/fonts" "/usr/local/lib/fonts"}

This example tells the server to look in the directories "/usr/lib/fonts" and "/usr/local/lib/fonts" when loading a font file.

I.3.10 Rx_Load_Bitmap

Function rx_load_bitmap {names}

names .... List of strings

Returns .. List of integers, new bitmap identifiers

Use: Reads a data file from disk and creates a bitmap (single plane pixmap) in memory. The command returns a unique integer identifier for this bitmap. The resulting bitmap may be used for tiling windows, masking graphics operations, etc.

Example:

BIT_MAP_ID := rx_load_bitmap "/us3/include/X11/bitmaps/dot"

This example loads the “dot” bitmap and returns an id for the resulting resource.

Notes: Unlike fonts, there is no search path for bitmap data files. The user must specify a full path when trying to load such a file.
I.3.11 Rx_Destroy_Bitmap

Procedure rx_destroy_bitmap {bids}
    bids ..... List of integer bitmap identifiers

Use: Deallocates the bitmaps given bids. This should only be used when a bitmap will not be needed again. Any further reference to the identifiers in bids will cause an error.

I.3.12 Rx_Load_Cursor

Function rx_load_cursor {names}
    names .... List of cursor names
    Returns .. List of integers, new cursor identifiers

Use: Creates a cursor that may be used for windows. The command returns a unique integer identifier for each cursor. The resulting cursor may be used for in any window.

Example:

CURSOR_ID := rx_load_bitmap {#XC_gumby #XC_hand1}

This example loads the “Gumby” and “Pointing hand” cursors returns ids for the resulting resources.

I.3.13 Rx_Destroy_Cursor

Procedure rx_destroy_cursor {cids}
    cids ..... List of integer cursor identifiers

Use: Deallocates the cursors given cids. This should only be used when a cursor will not be needed again. Any further reference to the identifiers in cids will cause an error.
I.4 Graphics

The commands in this section accept only pixel coordinates. See Class Specific Operations for more operations dealing with floating point coordinates. Except where otherwise specified, all of these commands operate on the default widget and use the default graphics context. With all of these commands, any output that would extend beyond a widget’s borders will automatically be clipped.

Warnings: The pixel coordinates used by X have their origin in the upper left hand corner of the widget. You must adjust accordingly.

I.4.1 Rx_Clear_Wid

Procedure rx_clear_wid {}

Use: This command clears the default widget.

I.4.2 Rx_Clear_Area

Procedure rx_clear_area {x, y, w, h, exp_flag}
  
  x ......... List of X coordinates
  y ......... List of Y coordinates
  w ......... List of widths
  h ......... List of heights
  exp_flag Boolean, generate an exposure event?

Use: Clears the specified rectangles within default widget. Specify an exp_flag of True to generate exposure events for the cleared regions.
I.4.3 Rx_Copy_Area

Procedure \texttt{rx\_copy\_area} \{\texttt{wsrc, wdest, xsrc, ysrc, w, h, xdest, ydest}\}

\begin{itemize}
  \item \texttt{src} \ldots Source widget identifier
  \item \texttt{xsrc} \ldots \textit{X} coordinate of rectangle to copy
  \item \texttt{ysrc} \ldots \textit{Y} coordinate of rectangle to copy
  \item \texttt{w} \ldots \ldots Width of rectangle
  \item \texttt{h} \ldots \ldots Height of rectangle
  \item \texttt{dst} \ldots Destination widget identifier
  \item \texttt{xdst} \ldots \textit{X} coordinate of destination
  \item \texttt{ydst} \ldots \textit{Y} coordinate of destination
\end{itemize}

Use: Copies the specified region of the source widget to the indicated point within the destination widget.

I.4.4 Rx_Draw_String

Procedure \texttt{rx\_draw\_string} \{\texttt{x, y, string}\}

\begin{itemize}
  \item \texttt{x} \ldots \ldots List of \textit{X} coordinates
  \item \texttt{y} \ldots \ldots List of \textit{Y} coordinates
  \item \texttt{string} \ldots Text strings to display
\end{itemize}

Use: Displays a text string. The upper left-hand corner of the first character is placed at \((x, y)\).

I.4.5 Rx_Str_Extent

Function \texttt{rx\_str\_extent} \{\texttt{string}\}

\begin{itemize}
  \item \texttt{string} \ldots Text strings to evaluate
\end{itemize}

Use: Computes the size, in pixels, of each string in \texttt{string}. The font in the default \texttt{gc} is used to compute these sizes.
I.4.6 Rx_Point

Procedure rx_point \{x, y\}

\begin{itemize}
\item x  \ldots  List of X coordinates
\item y  \ldots  List of Y coordinates
\end{itemize}

Use: Draw points given pixel coordinates. Any point that would fall outside the widget’s border will not be seen.

Notes: Currently, this command uses the XDrawPoint function – which is not producing visible points. This and rw_gfx_pnt must be modified to produce a small filled circle.

I.4.7 Rx_Line

Procedure rx_line \{x1, y1, x2, y2\}

\begin{itemize}
\item x1  \ldots  List of starting X coordinates
\item y1  \ldots  List of starting Y coordinates
\item x2  \ldots  List of ending X coordinates
\item y2  \ldots  List of ending Y coordinates
\end{itemize}

Use: Draw line segments given pixel coordinates. The parameter lists are combined to produce starting and ending (x,y) coordinate pairs.

I.4.8 Rx_Connected_Lines

Procedure rx_connected_lines \{x, y\}

\begin{itemize}
\item x  \ldots  List of X coordinates
\item y  \ldots  List of Y coordinates
\end{itemize}

Draws a single, multisegment line given a series of (x,y) coordinates. This command uses the \#default_drawmode default to determine how the coordinates are used. If \#default_drawmode is set to \#draw_absolute, then each coordinate is taken as offset from the origin, but if \#default_drawmode is set to \#draw_relative, then each coordinate is taken as an offset from the last point.
I.4.9 Rx_Arc

Procedure rx_arc {x, y, w, h, a1, a2}

- x ........ List of X coordinates of the upper left-hand corner of the bounding rectangles
- y ........ List of Y coordinates of same
- w ........ Widths of bounding rectangles
- h ........ Heights of bounding rectangles
- a1 ........ Starting angles, in 64ths of a degree, as measured from the three-o-clock position
- a2 ........ Extent of the arcs, in 64ths of a degree.

Use: Draws eliptical lines. The arcs drawn by this command correspond to those segments of an ellipse bounded by a box of dimension \( w \) and \( h \), having its upper left hand corner at \((x,y)\).

I.4.10 Rx_Fill_Arc

Procedure rx_fill_arc {x, y, w, h, a1, a2}

- x .......... List of X coordinates of the upper left-hand corner of the bounding rectangles
- y .......... List of Y coordinates of same
- w .......... Widths of bounding rectangles
- h .......... Heights of bounding rectangles
- a1 .......... Starting angles, in 64ths of a degree, as measured from the three-o-clock position
- a2 .......... Extent of the arcs, in 64ths of a degree.

Use: Functions the same as \texttt{rx.arc} above. Fills the resultant arc as specified by the current graphics context. Uses the \texttt{#GCArcMode} parameter to determine the boundaries of the fill region.

I.4.11 Rx_Circle

Procedure rx_circle {x, y, r}

- x .......... List of X coordinates
- y .......... List of Y coordinates
- r .......... List of Radii

Use: Draws circles having their center at \((x,y)\) and a radius of \(r\).
I.4.12  Rx_Fill_Circle

Procedure `rx_fill_circle` {x, y, r}
   x ........ List of X coordinates
   y ........ List of Y coordinates
   r ........ List of Radii

Use:    Functions the same as `rx_circle` above. Fills the resultant circle as specified
        by the current graphics context.

I.4.13  Rx_Rectangle

Procedure `rx_rectangle` {x, y, w, h}
   x ........ List of X coordinates
   y ........ List of Y coordinates
   w ........ List of widths
   h ........ List of heights

Use:    draws rectangles of height h and width w with upper right-hand corners at
        (x,y).

I.4.14  Rx_Fill_Rectangle

Procedure `rx_fill_rectangle` {x, y, w, h}
   x ........ List of X coordinates
   y ........ List of Y coordinates
   w ........ List of widths
   h ........ List of heights

Use:    Functions the same as `rx_rectangle` above. Fills the resultant box as specified
        by the current graphics context.
I.4.15  Rx_Fill_Polygon

Procedure rx_fill_polygon {x, y, type}
   x .......... List of X coordinates
   y .......... List of Y coordinates
   type ...... Single shape name

Use: Fills an arbitrary polygonal area of a widget. The (x,y) coordinates specify
a series of connected lines in the same manner as rx_connected_lines. The polygon en-
closed by these lines will be filled. If the first and last point are not the same, the poly-
gon will be automatically closed. As with rx_connected_lines, the #default_drawmode
default is used to determine how the coordinates should be interpreted. In addition,
the #GCFillRule parameter is used to resolve conflicts with self intersecting polygons.
The type parameter is a hint to the server about the layout of the polygon. This
parameter takes one of the following named values.
   #Convex ............... Paths may intersect
   #Nonconvex ............ No paths intersect, but not convex
   #Complex .............. Completely convex
I.5 Other/Internal

I.5.1 Rx_Bell

Procedure `rx_bell {}`

*Use:* Rings the keyboard bell.

I.5.2 Rx.Flush.Req

Procedure `rx_flush_req {}`

*Use:* Server flush. Sends all pending graphics output requests to the server.

*Notes:* See the Limitations section for details on usage.

I.5.3 Rx.Sync

Procedure `rx_sync {}`

*Use:* Synchronize server. See the section on Limitations for a discussion of this command.
Appendix J

Widget Summary

The following is a description of the available widget classes as well as some suggestions for their use.

If you do not find the information you need in this summary, there are several other good sources. Much of the information about the Athena widget set has been taken from *X Toolkit Widgets – C Language X Interface* by Ralph Swick and Terry Weissman. Documents covering any other widgets should also be available. In addition, widget header files and source code do not make easy reading, but lacking any of the above documents, it is usually possible to decipher the workings of a new widget from these files.

The situation might arise where you need a widget that cannot be constructed by using those available in this system, or you may wish to improve the performance of a certain common, but complex, widget structure. If such is the case, a new widget class can be written in C and added to this system. The structure of this interface has been tailored to permit such extensions in a relatively straightforward manner.

J.1 Common Resources

The following resources are recognized by most widget classes. In addition, a widget class can implement resources that are specific to widgets of that particular class or any of its subclasses.

**Resources:**

- “background” ............. Color id for a widget’s background
- “backgroundPixmap” ...... Pixmap id used to tile the widget background
- “borderColor” ............. Color id for a widget’s border
“borderPixmap” .......... Pixmap id used to tile the widget border
“borderWidth” .......... Width in pixels of a widget’s border
“cursor” ............... Cursor id for widget’s normal cursor
“foreground” .......... Color id for a widget’s foreground
“height” ............... Height in pixels of a widget
“mappedWhenManaged” ... Boolean value. If True, then a widget will appear on the screen when it is realized. If False, then a widget will be realized, but it must be manually unhidden afterwards.
“reverseVideo” .......... Boolean value that indicates whether foreground and background colors should be reversed for a widget.
“sensitive” ............. Boolean value. If True, then a widget will react to input normally. If False, the widget will not react to input and may appear stippled.
“width” ............... Width in pixels of a widget
“x” ................. X offset in pixels from the upper right hand corner of a widget’s parent
“y” ................. Y offset in pixels from the upper right hand corner of a widget’s parent
J.2 Common Events

The following events are available to instances of every widget class. To ask for notification for one of these events, use the event name as given below, but use the name \texttt{#xevent} in place of the widget class name. All events return at least one parameter, the id of the invoked notify.

Events:

\texttt{NoEvent} \hspace{1cm} \text{No events are dispatched to a widget.}

\texttt{KeyDown} \hspace{1cm} \text{Sent on a key press.}
\texttt{KeyUp} \hspace{1cm} \text{Sent on a key release. (Not all keyboards are able to generate these events)}

Handlers for the above keyboard events should accept two parameters. The first is an integer, the invoked notify’s id. The second is a string containing the character pressed or released (Possibly several characters if the keyboard has been rebound somehow)

\texttt{Btn1Motion} \hspace{1cm} \text{Sent on pointer motion with 1 depressed}
\texttt{Btn2Motion} \hspace{1cm} \text{Sent on pointer motion with 2 depressed}
\texttt{Btn3Motion} \hspace{1cm} \text{Sent on pointer motion with 3 depressed}
\texttt{Btn4Motion} \hspace{1cm} \text{Sent on pointer motion with 4 depressed}
\texttt{Btn5Motion} \hspace{1cm} \text{Sent on pointer motion with 5 depressed}
\texttt{BtnMotion} \hspace{1cm} \text{Sent on pointer motion with any button down}
\texttt{BtnUp} \hspace{1cm} \text{Sent on the release of any mouse button}
\texttt{BtnDown} \hspace{1cm} \text{Sent on the press of any mouse button}
\texttt{Motion} \hspace{1cm} \text{Sent on pointer motion}
\texttt{MotionHint} \hspace{1cm} \text{Sent on pointer motion}

Handlers for the above keyboard events should accept one parameter. This parameter is a list of five integers, the first one being the invoked notify’s id. Second and third are the X and Y pixel coordinates of the pointer within the widget. Fourth is the state of the mouse (clarify this).
The fifth integer is the button status.

#Enter .......... When the mouse enters the widget.
#Leave .......... When the mouse leaves the widget.

Handlers for the above two events should accept one parameter, the invoked notify’s id.

#Expose ........ Sent on when the contents of a widget have been destroyed and need refreshing. Handlers for the above keyboard events should accept one parameter. This parameter is a list of five integers, the first one being the invoked notify’s id. The next four integers describe the rectangular region within the widget that must be refreshed. These integers are the X and Y pixel coordinates of upper right hand corner, the width, and the height, respectively.

#StructNfty ....

#SubstructNfty .... Sent on When the widget has changed size or shape, No parameters are passed back because I am not quite sure what to pass back. Suggestions please – DTL
J.3 X Toolkit Intrinsic Widgets

J.3.1 Application Shell Widget Class

Name: #shell

Use: The shell widget class is a special kind of widget that is intended to be used solely as a top level widget. Instances of the shell class have no parent and may have only one child. A shell widget act as a sort of interface between the window manager and its one child.

Resources:

"iconName" ...............  
"iconPixmap" .............  
"iconWindow" ..........    
"iconMask" ..............  
"windowGroup" .........  
"saveUnder" ..........     
"transient" ..........    
"overrideRedirect" ......  
"allowShellResize" ......  
"createPopupChildProc" .. 
"title" .................  

J.3.2 Composite Widget Class

Name: #composite

Use: This is a rectangle on the screen that is allowed to have children. There are many subclasses of composite, each with a different flavor geometry manager. A geometry manager controls the placement of children within a composite-type widget. Widgets of the composite class have no geometry manager, so any children just stay where they are put, no matter what happens to the parent.
J.4 The Athena Widgets

J.4.1 Label Widget Class

Name: #label

Use: A label widget is simply a bit of text within a window. The text is limited to one line and may not be edited by the user. It may, however, be changed with the `rx_set_values` facility. The way the widget is designed, when the text string is changed, the widget automatically resizes to fit the new string.

Events:

#destroy .......... Sent when the instance has been destroyed. Handlers for this event should accept one parameter, the customary notify id.

Resources:

“label” ................. Text string to be displayed within the box.
“font” ................. Font id to be used for text.
“justify” ............... Position of the text string within the box. May be one of the following named values.
“internalHeight” ........ Padding distance in pixels between text and the top and bottom borders.
“internalWidth” ....... Padding distance in pixels between text and the left and right borders.
J.4.2 Command Widget Class

Name: #command

Use: The command widget is a subclass of label that has some behavior associated with it.

Events:
- **#destroy** ........... Sent when the instance has been destroyed. Handlers for this event should accept one parameter, the customary notify id.
- **#press** ............. Sent by the Notify() action. The default bindings attach this action to a rightmost button click within the widget. Handlers for this event should accept one parameter. the customary notify id.

Resources:
- “label” ................. Text string to be displayed within the box.
- “font” ................. Font id to be used for text.
- “justify” ............... Position of the text string within the box. May be one of the following named values.
- “internalHeight” ........ Padding distance in pixels between text and the top and bottom borders.
- “internalWidth” ........ Padding distance in pixels between text and the left and right borders.
- “highlightThickness” .... Thickness in pixels of the highlight border.
- “sensitive” ............. Boolean value indicating whether a widget will respond to user actions. If set to false, the widget will be stippled over and will not respond to any mouse events.

Actions:
- “notify” .................
- “set” .................
- “unset” .................
- “highlight” .............
“unhighlight” ..............

Default Bindings:

"<Btn1Down>" has_bindings "set()"
"<Btn1Up>" has_bindings "notify() unset()"
"<EnterWindow>" has_bindings "highlight()"
"<LeaveWindow>" has_bindings "unset(NoRedisplay) unhighlight()"
J.4.3 Scroll Widget Class

Name: #scroll

Use: The scrollbar widget is composed of a rectangular slide area with a contrasting slider (or thumb within. The bar can be oriented so that the slider moves either vertically or horizontally. The user generally manipulates the slider using the mouse keys and the application program may be informed of any changes by one of the special events implemented by this class.

Events:

#destroy ............ Sent when the instance has been destroyed. Handlers for this event should accept one parameter, the customary notify id.

#scroll ............. Sent by the NotifyScroll() action. By default, this action is bound to the release of any mouse button. The default bindings do not really support incremental scrolling, however, so this event is not currently of great use. Handlers for this event should accept one parameter. This parameter is a list of two integers, the invoked notify’s id, and a signed integer. The magnitude of this value is the position of the thumb in pixels relative to the top of the bar. The sign of this value indicates whether the pointer has moved forwards or backwards from its previous position.

#thumb ............. Sent by the NotifyThumb() action. By default, the NotifyThumb() action is bound to the middle mouse button. When this button is pressed, the slider is moved to the mouse’s current position. As longs as the button remains pressed, the slider will track the mouse and #thumb events will be repeatedly sent.

Handlers for this event should accept two parameters. This first is the usual invoked notify’s id. The second is a single floating point number indicating the position of the thumb as a percentage of the whole scrollbar.

Resources:
“length” .................. Length of a scrollbar in pixels. This is independent of the orientation of the bar.

“thickness” .............. Thickness of the scrollbar in pixels. This is independent of the orientation of the bar.

“orientation” ............ This is a name value indicating whether the scrollbar slider moves horizontally or vertically. One indicates a vertical slide and zero indicates a horizontal slide.

“scrollDownCursor” ...... Cursor to be used during scroll down with a vertical bar.

“scrollHorizontalCursor” .. Cursor to be used while a horizontal bar is at rest.

“scrollLeftCursor” ....... Cursor to be used during scroll left with a horizontal bar.

“scrollRightCursor” ........ Cursor to be used during scroll right with a horizontal bar.

“scrollUpCursor” ......... Cursor to be used during scroll up with a vertical bar.

“scrollVerticalCursor” ...... Cursor to be used while a vertical bar is at rest.

“shown” .................. Floating point percentage of scrollbar to be shaded.

“thumb” ................. Pixmap pattern to be used for the thumb.

“top” ..................... Floating point percentage position for the top of the thumb.

Actions:

“StartScroll” .............. Grabs the pointer and changes it to the appropriate cursor. This can take one of the three parameters “Forward”, “Continuous” and “Backward”.

“EndScroll” ............... Releases the pointer and resets it to the inactive cursor.

“MoveThumb” .............. Moves the slider to the current mouse position.

“NotifyThumb” ............ Sends back the #thumb event.

“NotifyScroll” ............ Sends back the #scroll event. It takes the parameters “Proportional” and “FullLength”. The FullLength parameter causes the notify to return the full length in pixels of the scrollbar, while
the Proportional parameter causes the offset in pixels from the top to be returned.

Default Bindings:

"<Btn1Down>" has bindings "StartScroll(Forward)"
"<Btn2Down>" has bindings "StartScroll(Continuous) MoveThumb() NotifyThumb()"
"<Btn3Down>" has bindings "StartScroll(Backward)"
"<Btn2Motion>" has bindings "MoveThumb() NotifyThumb()"
"<BtnUp>" has bindings "NotifyScroll(Proportional) EndScroll()"

Rx_Scrollbar_Set_Thumb

Procedure rx_scrollbar_set_thumb {wids, top, shown}

- wids ..... List of widget IDs
- top ..... List of reals, position of top
- shown ..... List of reals, percentage shown

Use: Sets the size and position of the slider within a scrollbar. wids specifies a number of scrollbar widgets. The shown parameter provides the size of the slider as a percentage of the whole and the top parameter provides the position of the slider top, also as a percentage of the whole.
J.4.4 String and Disk Text Widget Class

Name: #stringtext
Name: #disktext

Use: These widget classes are meant for viewing and editing of textual data. The stringtext class operates on a single string, while the disktext class operates on a disk file. These are both subclasses of the more general text widget class and therefore are identical except for the above differences. These classes are quite complex, with a large number of actions and a set of default bindings very similar to gnuemacs.

Resources:

“textOptions” ..............
“dialogHOffset” .......... Horizontal offset in pixels of “Insert File” dialog box
“dialogVOffset” .......... Vertical offset in pixels of “Insert File” dialog box
“displayPosition” .........
“insertPosition” ..........
“leftMargin” .............
“selectTypes” ............
“selection” ..............
“editType” ............... The editing mode. Can be one of the following name values.
    #textRead View without changing
    #textAppend Add to existing text
    #textEdit Full editing permission
“file” ..................... File name for a disktext class widget
“string” ................... String for a stringtext class widget
“length” .................. Current string length for a stringtext class widget
“maxLength” .............. Maximum string length for a stringtext class widget
Actions:

“forward-character” ........
“backward-character” ......
“forward-word” .......... 
“backward-word” ........
“forward-paragraph” ......
“backward-paragraph” ..... 
“beginning-of-line” ........
“end-of-line” .............
“next-line” ............. 
“previous-line” .......... 
“next-page” ............. 
“previous-page” ........ 
“beginning-of-file” .......
“end-of-file” .............
“scroll-one-line-up” ........
“scroll-one-line-down” .... 
“delete-next-character” ..... 
“delete-previous-character” .
“delete-next-word” ........
“delete-previous-word” ....
“delete-selection” .......
“kill-word” ............. 
“backward-kill-word” ...... 
“kill-selection” ........ 
“kill-to-end-of-line” ......
“kill-to-end-of-paragraph” ..
“unkill” ................. 
“stuff” .................
“newline-and-indent” ......
“newline-and-backup” ..... 
“newline” ............... 
“select-word” ........... 
“select-all” ............ 
“select-start” .......... 
“select-adjust” .......... 
“select-end” ............
“extend-start” .......... 
“extend-adjust” .......... 
“extend-end” ............
“redraw-display” .......... 
“insert-file” ............
APPENDIX D. USERS GUIDE – Widget Summary

“insert-char”  ....................
“focus-in”  ........................
“focus-out”  .........................

Default Bindings:

"Ctrl<Key>F" has bindings "forward-character()"
"<Key>0xff53" has bindings "forward-character()"
"Ctrl<Key>B" has bindings "backward-character()"
"<Key>0xff51" has bindings "backward-character()"
"Meta<Key>F" has bindings "forward-word()"
"Meta<Key>B" has bindings "backward-word()"
"Meta<Key>]\" has bindings "forward-paragraph()"
"Ctrl<Key>[" has bindings "backward-paragraph()"
"Ctrl<Key>A" has bindings "beginning-of-line()"
"Ctrl<Key>E" has bindings "end-of-line()"
"Ctrl<Key>N" has bindings "next-line()"
"<Key>0xff54" has bindings "next-line()"
"Ctrl<Key>P" has bindings "previous-line()"
"<Key>0xff52" has bindings "previous-line()"
"Ctrl<Key>V" has bindings "next-page()"
"Meta<Key>V" has bindings "previous-page()"
"Meta<Key>\]\" has bindings "beginning-of-file()"
"Meta<Key>\]\]\" has bindings "end-of-file()"
"Ctrl<Key>\Z\" has bindings "scroll-one-line-up()"
"Meta<Key>\Z\Z\" has bindings "scroll-one-line-down()"
"Ctrl<Key>\D\" has bindings "delete-next-character()"
"Ctrl<Key>\H\" has bindings "delete-previous-character()"
"<Key>0xff7f" has bindings "delete-previous-character()"
"<Key>0xffff" has bindings "delete-previous-character()"
"<Key>0xff08" has bindings "delete-previous-character()"
"Meta<Key>\D\" has bindings "delete-next-word()"
"Meta<Key>\H\" has bindings "delete-previous-word()"
"ShiftMeta<Key>\D\" has bindings "kill-word()"
"ShiftMeta<Key>\H\" has bindings "backward-kill-word()"
"Ctrl<Key>\W\" has bindings "kill-selection()"
"Ctrl<Key>\K\" has bindings "kill-to-end-of-line()"
"Meta<Key>\K\" has bindings "kill-to-end-of-paragraph()"
"Ctrl<Key>\Y\" has bindings "unkill()"
"Meta<Key>\Y\" has bindings "stuff()"
"Ctrl<Key>\J\" has bindings "newline-and-indent()"
"<Key>0xff0a" has bindings "newline-and-indent()"
"Ctrl<Key>O" has bindings "newline-and-backup()"
"Ctrl<Key>M" has bindings "newline()"
"<Key>0xff0d" has bindings "newline()"
"Ctrl<Key>L" has bindings "redraw-display()"
"Meta<Key>I" has bindings "insert-file()"
"<FocusIn>" has bindings "focus-in()"
"<FocusOut>" has bindings "focus-out()"
"<Btn1Down>" has bindings "select-start()"
"Button1<PtrMoved>" has bindings "extend-adjust()"
"<Btn1Up>" has bindings "extend-end()"
"<Btn2Down>" has bindings "stuff()"
"<Btn3Down>" has bindings "extend-start()"
"Button3<PtrMoved>" has bindings "extend-adjust()"
"<Btn3Up>" has bindings "extend-end()"
"<Key>" has bindings "insert-char()"
"Shift<Key>" has bindings "insert-char()"
J.4.5 Button Box Widget Class

Name: #box

Use: The box widget class is a subclass of composite with a geometry manager intended for organizing button-type widgets. The geometry manager tries to order all children along the upper left portion of the widget.

Resources:

“hSpace” ................. Distance in pixels to leave around each child’s left and right borders.

“vSpace” ................. Distance in pixels to leave around each child’s top and bottom borders.
J.4.6 Form Widget Class

Name:  #form

Use:  The form widget class is a subclass of composite with a rather flexible geometry manager. The following resource is applicable to all instances of the form widget class.

Resources:

“defaultDistance”  . . . . . . . . . . Length in pixels to be used as a default for “horizDistance” and “vertDistance”. (see below)

When children are added to a form widget, they may specify where and how they want to be placed. As such, the following resources are not used for the form widget itself, but instead may be set for each child widget.

Resources:

“top”  . . . . . . . . . . . . . . . . . . . . Indicates how to position the child when the form is resized. Takes one of the following name values:

#ChainTop Maintains a constant distance from the top of the form
#Rubber Maintains a proportional distance from the top of the form

“bottom”  . . . . . . . . . . . . . . . . . . . Indicates how to position the child when the form is resized. Takes one of the following name values:

#ChainBottom Maintains a constant distance from the bottom of the form
#Rubber Maintains a proportional distance from the bottom of the form

“left”  . . . . . . . . . . . . . . . . . . Indicates how to position the child when the form is resized. Takes one of the following name values:

#ChainLeft Maintains a constant distance from the left edge of the form
#Rubber Maintains a proportional distance from the left edge of the form

“right” ................. Indicates how to position the child when the form is resized. Takes one of the following name values:

#ChainRight Maintains a constant distance from the right edge of the form

#Rubber Maintains a proportional distance from the right edge of the form

“horizDistance” .......... Length in pixels between this child and any widgets to either side.

“fromHoriz” .............. Widget identifier. If not null, this child is placed to the immediate right (separated by “horizDistance”) of the specified widget. If null, the child is placed along the left border of the form.

“vertDistance” .......... Length in pixels between this child and any widgets above or below.

“fromVert” ............... Widget identifier. If not null, this child is placed immediately below (separated by “vertDistance”) the specified widget. If null, the child is placed along the top border of the form.

“resizable” .............. Boolean value, indicates whether this child is allowed to resize. Normally False.
J.4.7 Vertical Pane Widget Class

Name: #vpane

Use: The vpane widget class is another subclass of composite with a specialized geometry manager. This widget organizes its children in a vertical fashion. Each child forms a pane that extends the width of the parent. On the borders between children a small knob appears. This knob is called a grip and can be used to move the borderline that it falls on. This has the effect of shrinking one widget while enlarging another. The following resource is applicable to instances of the vpane class:

Resources:

“gripIndent” ............... Indent in pixels from the left border for the grip widgets.

As with the form widget, when children are added to a vpane widget, they may specify how they want to be handled. As such, the following resources are not used for the vpane widget itself, but instead may be set for each child widget.

Resources:

“allowResize” .............. Boolean value, normally False, that indicates whether this child is allowed to make resize requests.

“min” ..................... Minimum height of this child in pixels.

“max” ..................... Minimum height of this child in pixels.

“skipAdjust” .............. Boolean value, normally False, that indicates whether the vpane widget is allowed to automatically resize this child.
J.4.8 Viewport Widget Class

Name:  #viewport

Use:  The viewport widget class is a special subclass of composite that is intended to manage a single child widget. This child is generally larger than the viewport and may be moved with optional scrollbars. Scrollbars are automatically provided if the child widget is larger than the viewport.

Resources:

“allowHoriz” ............... Boolean value, normally false, indicating whether horizontal scroll bars are allowed.

“allowVert” ............... Boolean value, normally false, indicating whether vertical scroll bars are allowed.

“forceBars” ............... Boolean value, normally false, indicating whether scrollbars should be displayed even if they are not needed.

“useBottom” ............... Boolean value. If false, a scrollbar is displayed along the viewport’s top edge. If true, it is displayed along the bottom edge.

“useRight” ............... Boolean value. If false, a scrollbar is displayed along the viewport’s left edge. If true, it is displayed along the right edge.
J.4.9 Dialog Box Widget Class

Name: #dialog

Use: The dialog class widgets are compound widgets that can be used to prompt a user for a string of input and then use one of several buttons to confirm the input. A dialog widget is divided into three lines, the first of which contains a text label for the box. The second line is a text entry box, perhaps containing some default text. The third line may contain command buttons that the user can use to control the behavior of the box. The label and text entry boxes are created automatically. To add buttons, simply create them as children of the dialog box.

Resources:

“grabFocus” ............... Unknown
“label” .................... Text label for the dialog box.
“maximumLength” ........ Maximum length of the input text.
“value” .................... Default text to place in the text entry box.

Rx_Dialog_Get_String

Function rx_dialog_get_string {wid}

  wids ..... List of widget IDs
  Returns .. List of strings

Use: Retrieves the user input string associated with each dialog widget. The wid argument specifies the dialog boxes to retrieve from.
J.5 Locally Produced Widgets

J.5.1 Graphics Widget Class

Name: \#gfx

Use: This mapped graphics widget is a subclass of composite with several mapping functions associated with it. It should be used for mapping graphics produced in some world coordinate system to the pixel coordinates used by X. The associated transforms are meant to insulate the user from the fact that he is working in the X graphics environment.

Resources:

“virX” .................... X origin of window in floating point world coordinates

“virY” .................... Y origin of window in floating point world coordinates

“virHeight” ............... Height of window in floating point world coordinates

“virWidth” ............... Width of window in floating point world coordinates

“resizeMode” ............. Controls how the scale is controlled when the widget is resized. Not yet implemented.
**Rw_Gfx_Lne**

**Procedure** `rw_gfx_lne {x1, y1, x2, y2}`

- `x1` ........ List of starting X coordinates
- `y1` ........ List of starting Y coordinates
- `x2` ........ List of ending X coordinates
- `y2` ........ List of ending Y coordinates

**Use:** Draw lines given floating point world coordinates. These world coordinates are mapped through the widget instance to actual screen coordinates. This operation may only be used on instances of the `gfx` widget class. As usual, any line that would extend beyond the widget’s border will be clipped. The parameter lists are combined to produce starting and ending (x,y) coordinate pairs.

**Notes:** This command relies on the default widget (which must be of class `gfx`) and default GC.

**Rw_Gfx_Pnt**

**Procedure** `rw_gfx_pnt {x, y}`

- `x` ........ List of X coordinates
- `y` ........ List of Y coordinates

**Use:** Draw points given floating point world coordinates. These world coordinates are mapped through the widget instance to actual screen coordinates. This operation may only be used on instances of the `gfx` widget class. As usual, any point that would fall outside the widget’s border will not be seen. The parameter lists are combined to produce (x,y) coordinate pairs.

**Notes:** This command relies on the default widget (which must be of class `gfx`) and default GC. Currently, this command uses the `XDrawPoint` utility – which is not producing visible points. This and `rx_pnt` must be modified to produce a small filled circle.
**Rw_Gfx_Locator**

**Function** rw_gfx_locator {}  
Returns .. Two reals, \((x, y)\) coordinate pair

**Use:** Mouse query. This command blocks execution until a mouse button is pressed in the default widget. For the time being, this widget must be an instance of the gfx class. The coordinates returned are a floating point pair representing an \((x, y)\) coordinate in this widget’s virtual coordinates.
Appendix K

X/ROSE Reference Card
APPENDIX D. USERS GUIDE – Widget Summary

X/Rose Reference Card

System Procedures
UNIX commands
Procedure define_path (NAME, PATH) Function enter () Procedure system (S:STRING)

Widget commands
Function rs_create_widget (PARENT,CLASS,VALUE) Procedure rs_destroy_widget (WID) Procedure rs_realize_tree (WID) Procedure rs_hide (WID) Procedure rs_unhide (WID) Procedure rs_popup (SPEC) Procedure rs_popup_relative (SPEC,VID,X,Y) Procedure rs_popupdown (WID) Procedure rs_grab (SPEC) Procedure rs_ungrab (VID) Procedure rs_set_defaults (VALUE) Procedure rs_set_values (VID,VALUE) Function rs_get_values (VID,VALUE) Procedure rs_set_bindings (VID,BINDINGS) Procedure rs_add_bindings (VID,BINDINGS) Procedure rs_set_sensitive (SEBS)

Notification Commands
Function rs_create_notify (VID,CLASS,EVENT,PROC) Procedure rs_destroy_notify (VID) Procedure rs_start_notify (VID) Procedure rs_stop_notify (VID) Procedure rs_change_notify (PROC) Procedure rs_change_notify_values (VID) Procedure rs_change_notify_event (VID,CLASS,EVENTS)

Resource & Utility Commands
Function rs_create_gc (GC,VALUES) Procedure rs_set_gc (GC,GCVALUES) Procedure rs_copy_gc (SRC,GOT,VALUES) Function rs_load_font (FONTSNAME) Procedure rs_destroy_font (FID) Function rs_set_font (PATHS) Function rs_set_font_path () Procedure rs_set_font_fields (PATHS) Function rs_change_font_field (PATHS) Function rs_load_cursor (NAME) Procedure rs_destroy_cursor (CID)

Graphics Commands

Other Procedures
Procedure rs_flush_req () Procedure rs_sync () Procedure rs_bell ()

Functions & Mappings
with_path = use, readf, write
has_binding = rs_set_bindings, rs_add_bindings
has_sensitive = rs_set_sensitive
has_grab = rs_popup, rs_popup_relative
has_type = rs_set_values
has_value = rs_set_values, rs_set_defaults, rs_create_widget
affl_name = has_value
affl = has_value
has_gc_value = rs_set.gc, rs_set.gc
resource has_type type name

X/UI Types
#KeyDown #KeyUp
#BtmDown #BtmUp
#Enter #Leave
#Motion #MotionHint
#Btn1Motion #Btn2Motion
#Btn3Motion #Btn4Motion
#Btn5Motion #Btn6Motion
#Expose #StructNtfy

System Defaults
Named parameters accepted by the Set Defaults function
#defaultwid #dflwid
#defaultgc #dfl gc
#defaultdrawmode #dfldrawmode
#defaultonlyfmt #dflonlyfmt

Values for the #defaultdrawmode parameter
#drawAbsolute #drawRelative

Graphics Context
Named parameters accepted by the various Graphics Context functions.
#GCFill
#GCPalette
#GCFont
#GCForeground
#GCSecondary
#GCSubwindow
#GCstroke

Values for the #GCFill parameter
#GCFill #GCPalette

Values for the #GCFont parameter
#GCFont

Values for the #GCForeground parameter
#GCForeground

Values for the #GCSecondary parameter
#GCSecondary

Values for the #GCSubwindow parameter
#GCSubwindow

Values for the #GCstroke parameter
#GCstroke
#LineSolid
#LineOnOffDash
#LineDoubleDash

Values for the #GCCapStyle parameter
#CapNotLast
#CapButt
#CapRound
#CapProjecting

Values for #GCCapStyle parameter
#JoinMiter
#JoinRound
#JoinBevel

Values for #GCFillStyle parameter
#FillSolid
#FillTiled
#FillStippled
#FillOpaqueStippled

Values for #GCFillRule parameter
#EvenOddRule
#WindingRule

Values for #GCSubwindowMode parameter
#ClipByChildren
#IncludeInferiors

Values for #GCArcMode parameter
#ArcChord
#ArcPieSlice

Available Cursors

#XC_num_glyphs  #XC_Cursor
#XC_arrowswim  #XC_Cursor
#XC_based_arrow\_up  #XC_XCursor
#XC_bogosity  #XC_bottom\_left\_corner
#XC_bottom\_right\_corner  #XC_bottom\_side
#XC_bottom\_tie  #XC_box\_spiral
#XC_center\_ptr  #XC_circle
#XC_cross  #XC_cross\_reverse
#XC_crosshair  #XC\_diamond\_cross
#XC\_left\_side  #XC\_left\_tee
#XC\_leftbutton  #XC\_ll\_angle
#XC\_middle\_button  #XC\_man
#XC\_pencil  #XC\_grip
#XC\_plus  #XC\_question\_arrow
#XC\_right\_ptr  #XC\_right\_side
#XC\_right\_tee  #XC\_right\_button
#XC\_sb\_logo  #XC\_sb\_down\_arrow
#XC\_sb\_double\_arrow  #XC\_sb\_left\_arrow
#XC\_sb\_right\_arrow  #XC\_sb\_up\_arrow
#XC\_sb\_v\_double\_arrow  #XC\_shuttle
#XC\_sizing  #XC\_spider
#XC\_spraycan  #XC\_star
#XC\_target  #XC\_cross
#XC\_tt\_left\_arrow  #XC\_tt\_left\_corner
#XC\_tt\_right\_corner  #XC\_tt\_side
#XC\_tt\_tee  #XC\_trek
#XC\_u\_angle  #XC\_umbrella
#XC\_u\_angle  #XC\_watch
#XC\_xterm

Misc. System Constants

Grab Types for Rx\_Popup
#GrabNonExclusive
#GrabExclusive
#GrabSpringLoad