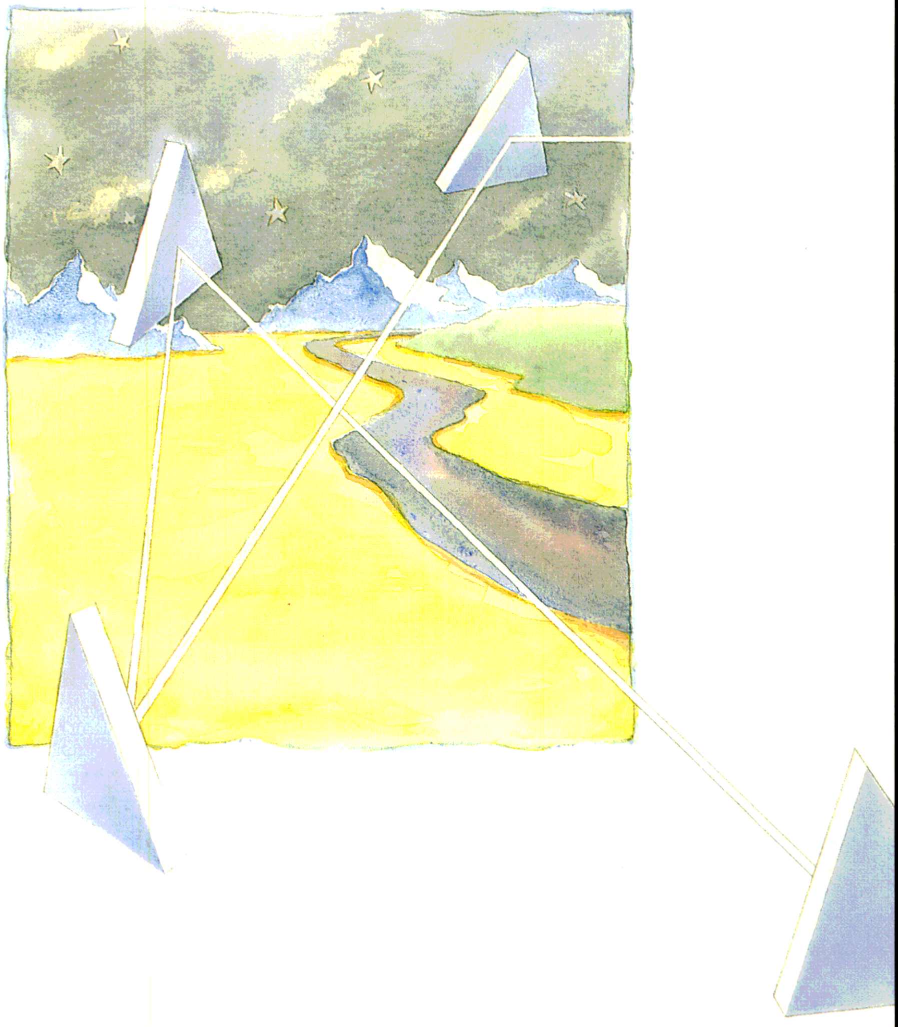


Using the X Window System on Apollo Workstations

apollo

Using the X Window System on Apollo Workstations

Order No. 015213-A02

First Printing: December, 1988
Latest Printing: October, 1989

10 9 8 7 6 5 4 3 2 1

Preface

This manual describes the Apollo Domain/X11 implementation of the Version 11 protocol of the MIT X Window System. This manual explains how to

- Start **Xapollo**, Apollo's share-mode X server.
- Run X clients locally or over a network.
- Compile X clients and link them to Xlib (and other libraries).
- Write X clients that use Apollo native graphics.

Refer to the appropriate file in **/install/doc/apollo** for information regarding product support.

Is this Manual for You?

If you're already an experienced X programmer or otherwise used to a UNIX environment running the X Window System, and you wonder about Apollo's implementation of X, then you should find most of what you need in Chapters 4 and 5 and the Appendixes.

We assume that you're a casual or new X user, but an experienced programmer or UNIX user. If you develop applications using X, the X Toolkit, or other toolkits, or if you code applications for which X must be running, then this manual will help you set up your X environment, but it will not show you how to program with Xlib or the X Toolkit (for that, see the "Related Documentation" section in this preface).

If you want to run Open Dialogue or other X-based programs, we'll tell you how to configure and run X at boot or login time, and how to run X-based programs. You may learn a little about UNIX if it's new to you. You should be able to use a text editor, and you may need a system administrator to issue some of the commands.

How to Read this Manual

We've organized this manual as follows:

Chapter 1	Introduces the X Window System, the **Xapollo** server, and popular X clients.
Chapter 2	Tells how to run the **Xapollo** server and the **uwm** window manger.
Chapter 3	Describes how to set environment variables and how to run many of the X clients.
Chapter 4	Tells how to modify the keyboard using **xmodmap**. Also describes the server's implementation of color and fonts.
Chapter 5	Describes how to compile and link X client programs and how to run Apollo native graphics in an X window.
Appendixes	Provide troubleshooting hints, a checklist for running **Xapollo** and X clients, and code for a native-graphics X client.
Glossary	Terms you need to know if you're new to X or not familiar with UNIX.

Online Files

This section lists the major directories that contain library and include files, online man pages, release notes, and other related documentation. For a complete list of directories, see the release notes.

Domain/X11 software physically resides in a series of directories under **/usr/X11**. These directories are, in turn, pointed to by a series of links under /usr (e.g., **/usr/include/X11**, **/usr/lib/X11**). The links provide proper functioning of both the UNIX and MIT standard pathnames. The links are described further in the release document; this manual uses the link names and not the files' actual locations.

Libraries and Include Files

The files of most importance in writing code for X clients are discussed in Section 5.2 (in Chapter 5) and are in the following directories:

- **/usr/include/X11** contains all the header files.
- **/usr/lib/X11** contains X libraries and various data files such as the RGB color database and the X fonts.
- **/lib** contains run–time libraries **x11lib**, **xawlib**, **xtlib**, and **x11paslib**.

Sample Source and Binary Files

Sample source code and a binary file demonstrating how to run Apollo native graphics in an X window (as described in Section 5.4 in Chapter 5) are in **/domain_examples/X11**. (We print the source for one of these in Appendix D.)

Sources for the examples in O'Reilly, Vol. 3, *X Window System User's Guide* (see "Related Documentation" in the following section in this Preface) are in /usr/src/X11, in a .Z file that your system administrator might have uncompressed for you.

Online Documentation

The Domain/X11 release contains a number of text files that provide further documentation for the product. They can be found in

- The **man** pages that are accessed online through the **man** command.

- **/usr/src/X11**, which contains nroff/troff versions of the MIT documents, if installed by the system administrator. These documents may be in archived form.

Related Documentation

Apollo has licensed several books to redistribute to customers. You can order them as well as Apollo documentation by calling (from within the United States) **Apollo Direct** at **1-800-225-5290**.

O'Reilly & Associates' References

The references licensed from O'Reilly & Associates include:

- *X Protocol Reference Manual*, Vol 0 (017140), describing the communication protocol used for connecting applications to workstations: the contents of each type of message that passes between the application and the workstation, and what effects each message has.

- *Xlib Programming Manual*, Vol. 1 (011241), covering general X program structure, creating windows, window attributes, graphics contexts, the display of graphics and text, color, event handling, keyboard and pointer control, and interclient communication conventions. (Errata and examples are in **/usr/src/X11/oreilly**.)

- *Xlib Reference Manual*, Vol. 2 (013418), listing the Xlib calls alphabetically, with appendixes organized in functional groups (calls about keyboards, windows, etc.), including macros, the color database, events, symbols, fonts, and so on. (Errata and examples are in **/usr/src/X11/oreilly**.)

- *X Window System User's Guide*, Vol. 3 (015534), describing X Window System concepts and the common clients, including an especially good discussion of the **.Xdefaults** and other resource files.

- *X Toolkit Programming Manual*, Vol. 4 (017131), providing concepts and examples that show how to program with the various X Toolkit routines to use and write widgets.

- *X Toolkit Intrinsics Reference Manual*, Vol. 5 (017132), providing a complete programmer's reference for X Toolkit Intrinsics, including reference pages for Intrinsics, required widget types, Athena widgets, etc.

- *X Window System Quick Reference* (017141), providing an alphabetical listing of the Xlib and Toolkit functions and their arguments.

Other Reference Materials

The bibliography for X keeps expanding, and the various references keep changing from first to second editions as the X "industry" better understands such issues as interclient communications, the widget sets that are most popular, the new releases from MIT, the impact of OSF/Motif, and so on. Keeping up with these changes in the next year or so will continue to be an important part of learning to use and program with X.

Another excellent, licensed reference is *Introduction to the X Window System* (017133), by Oliver Jones (Prentice Hall, 1989). Directed more at experienced programmers, it explains how to understand and use Xlib (concepts, windows, graphics, text, color, images, mouse, keyboard, event-handling, and information-sharing).

One of the more popular books on writing widgets is *The X Window System: Applications and Programming with Xt* by Doug Young (Prentice Hall, 1989), describing and demonstrating how to write applications using both the X Toolkit layer (Xt Intrinsics and widgets) and the lower-level Xlib C interface. The book also discusses how to extend the X Toolkit by writing new widgets. Look for a second edition containing the OSF/Motif widget set.

Man Pages and MIT Documentation

Other than the manual that you are reading, most of the information on how to be a user of X is in the online X man pages. You can read them by issuing the command

```
man page_name
```

These man pages include descriptions of X clients such as **xterm**, **uwm**, **xclock**, **xedit**, etc. Each page describes the client's or command's purpose, any command line options, and any internal commands. The most important are listed in Table P-1; a complete list of client names is in Table 3-2.

Table P-1. Important Man Pages

Man Pages	Purposes
Xapollo	Describes the options that establish either X or the DM as the primary window system, and how to configure the keyboard.
Xserver	Describes the command line settings common to all X servers.
X	Provides short descriptions of command line options that are common to all X clients. For example, it describes how to give a client an X "geometry"—the X word for specifying the size and location of the client when you start it.

X comes with a great deal of documentation from MIT. If your system administrator installed it, you should find the documentation sources in nroff/troff format in /usr/src/X11. (Ask your system administrator to uncompress or un-archive these files if your directory contains only a **README**, a **doc.tar.Z**, and an **oreilly.tar**—or read the **README** file and follow its directions.)

The major MIT documents include

- *X Window System, Version 11 Release 3 Release Notes*. Describes release changes and directories.
- *Xlib–C Language X Interface*. Describes the Xlib interface. This is the original MIT Xlib reference manual with limited explanatory material.
- *Inter–Client Communication Conventions Manual*. Describes which functions are proper for window managers, and which for clients. This is the most up–to–date version available at the time of this release of Domain/X11.
- *A Simple X11 Client Program (Hello World)*. Shows how a "well–behaved" X client should initialize itself.
- *X Window System Protocol*. Defines the X protocol. Its discussion of keyboards and its keysym encoding appendix are especially pertinent.

The MIT X Toolkit, also on the tape, contains these documents:

- *X Toolkit Widgets—C Language X Interface*
- *X Toolkit Intrinsics—C Language X Interface*
- *How to Write a Widget*

Other Apollo Documentation

This manual assumes that you have some experience with UNIX, Apollo operating systems, and the SysV or Berkeley BSD environments, but if you don't, you may find some of the following manuals useful:

- *Making the Transition to SR10 Operating System Releases* (011435)—if you're familiar with earlier releases.
- *Getting Started with Domain/OS* (002348)—the beginner's guide to using BSD software on an Apollo node.
- *Using Your BSD Environment* (011020)—if you're unfamiliar with this environment. Its preface suggests some basic introductions to UNIX.

- *Configuring and Managing TCP/IP* (008543)—if you configure or administer TCP/IP database files.

- *Managing System Software*—these are three different manuals: Aegis (010852), BSD (010853), and SysV (010851).

The file **/install/doc/apollo/os.v.10.***x*__**manuals** lists current titles and revisions for all available Apollo manuals. You may also use this file to check that you have ordered all of the manuals that you need. (If you are using the Aegis environment, you can access the same information by typing **help manuals.**)

Refer to *Apollo Documentation Quick Reference* (002685) and *Domain Documentation Master Index* (011242) for lists of documents.

Summary of Technical Changes

Many of the principal changes from the V1.0 version that runs on SR9.7 include new system files, a new location for the server and server log, a general change of paths from /sys/x11 to **/usr/X11**, support of UDS (UNIX Domain Sockets), an updated termcap, more efficient use of disk space, new run-time libraries (one with globally-recognized symbols) allowing for smaller clients, support for starting the server at boot time; and Makefiles for all clients.

The changes from the V1.1 version principally include

- The use of Release 3 libraries and clients rather than Release 2

- The preferred use of **xinit** (via the **startx** script) to start the X server

- The removal of the **xclient** client, the **X.starterkit** directory, the Makefiles for clients, and the 3D GMR and Open Dialogue example programs

- Support for the X Toolkit

- Improved performance

Does This Manual Support Your Software?

To verify which version of operating system software you are running, type

```
bldt
```

Check in **/install/doc/apollo** for the release documents whose names end in **__notes** to determine which version of Domain/X11 you should be using. Read Chapter 3 of the file **os.v.10.*x*__notes**, or Chapter 1 of the file named **domain_x11.v.*x.y*__notes**. (Check with your system administrator if you cannot find these.)

You can also

- Telephone **1-800-225-5290**. If you are calling from outside the U.S., call us at **(508) 256-6600** and ask for **Apollo Direct Channel**.
- Refer to the lists of manuals described in a previous section, "Other Apollo Documentation."

You can tell which of two versions of the same manual is newer, by looking at the order number (on the title page). Every order number ends with an "A" followed by two digits, for example, **-A00**. A higher A number is generally used for a newer manual. For instance, **-A02** indicates a newer manual than **-A01**.

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. To make it easy for you to communicate with us, we provide the Apollo Product Reporting (APR) system for comments related to hardware, software, and documentation. By using this formal channel, you make it easy for us to respond to your comments.

You can get more information about how to submit an APR by consulting the appropriate Command Reference manual for your environment (Aegis, BSD, or SysV). Refer to the **mkapr** (make Apollo product report) shell command description. You can view the same description online by typing

```
$ man mkapr (in the SysV environment)
% man mkapr (in the BSD environment)
$ help mkapr (in the Aegis environment)
```

Or, you may use the Reader's Response Form at the back of this manual to submit comments about the manual.

Documentation Conventions

commands/keywords	Bold words or characters in formats and command descriptions represent commands or keywords that you must use literally. Bold words in text indicate the first use of a new term. Filenames and pathnames are also in bold.
user–supplied values	Italic words or characters in formats and command descriptions represent values that you must supply.
sample user input	Commands that you can enter appear in bold and/or blue typeface.
`output/source code`	Program code, and lines in files and system displays, appear in typescript.
[ ]	Square brackets enclose optional items in formats and command descriptions.
{ }	Braces enclose a list from which you must choose an item.
|	A vertical bar separates items in a list of choices.
< >	Angle brackets enclose the name of a key on the keyboard, as in <RETURN>.

	When a key is discussed in its X sense or as a keysym, we capitalize just the first letter and bold the name, as in "A modifier key is a key like **Shift** or **Control** that can modify the meaning of a key event."
^	The circumflex followed by the name of a key indicates a control character sequence. Hold down <CTRL> while you press the key.
\	A backslash indicates that what we print as two or more physical lines should be placed together on one line in your file or command line.
⋮	Vertical ellipsis points indicate the omission of irrelevant parts of a figure or example.
——— ⊞ ———	This symbol indicates the end of a chapter or part of a manual.

——— ⊞ ———

Contents

Chapter 4 Keyboards, Workstations, Resources

Figures

Tables

Chapter 1

Introduction

Contents

Chapter 1

Introduction

This chapter introduces the X Window System and describes the **Xapollo** server that runs in "share mode" with the Apollo Display Manager (DM). **Share mode** lets you display DM windows on the same screen as X client programs such as **xterm** and **xclock**. You can choose either the DM or X as your primary window system and use either the DM or an X window manager such as **uwm**.

X runs with Sys5.3, BSD4.3, and Domain/OS (Aegis). We'll start out using commands suitable for all of these environments, but most of this manual is written with an emphasis on BSD. We define new terms and concepts as we go along and also in the glossary.

If you get error messages, look at the brief troubleshooting guide in Appendix B or the checklist in Appendix A, and read Chapter 2 (especially Section 2.3).

1.1 Starting Quickly

You can check if **Xapollo** is already running by typing

```
ps -ax                    (for BSD shells)
```

or

```
ps -e                         (for SysV shells)

or

pst                           (for Aegis shells)
```

and looking for the name **Xapollo** in the resulting list of processes. If it's there, you may want to read Section 2.5 and Chapter 3 for directions on running window managers and clients. If you don't find it, you have several options, discussed below.

1.1.1 The Fast Track at Boot Time

Xapollo may be installed automatically in your **/etc/daemons** directory as described in this section. In case it isn't, here's how to do it.

It's simplest to treat **Xapollo** as a built-in server/daemon and run it at boot time. We install a file named **/etc/rc** with the appropriate command lines to run **Xapollo**, provided that a file named **Xapollo** exists in your **/etc/daemons** directory.

To add **Xapollo** to your **/etc/daemons** directory, enter the command

```
touch /etc/daemons/Xapollo          (for UNIX)

   or

/com/crf /etc/daemons/Xapollo       (for Domain/OS)
```

Reboot with the DM **shut** command and use **ex domain_os** to recover. (If you're logged in as root, you can use **/etc/reboot**.)

Now you can take several reading paths through the manual:

- You've started the server in a DM-owned root (see Section 1.5.1). See Sections 2.3.2 and 4.1.1 for more explanation, particularly about the keyboard configuration file.
- To read about the **/etc/rc** file, see Sections 2.2.2 and 2.2.3.
- For information on **xmodmap** and more keyboard configuration, see Section 4.2.

- To read about X resource files, see Section 4.4.
- For environment variables, see Section 3.1.

1.1.2 The Middle Lane at Login

If you want **Xapollo** to start at login, just enter the following command in one of your startup scripts (for example, **.login**):

```
/usr/bin/X11/startx
```

The **startx** script will start **Xapollo** and an **xterm**. When you exit (e.g., ^D) from the **xterm**, **Xapollo** will go away, too. See Section 2.2.4 for more information on the **startx** script and Section 3.1.1 for more information about appropriate startup script locations.

1.1.3 The Scenic Route at Run Time

You can also run **startx** after you've logged in. However, as a teaching device, we'll give you the command lines that you can use to start up the server, run the **xclock** client (Figure 1-1), and then stop them both. This is "scenic" because it takes time to understand the **startx** script and its use of **xinit** and the X resources files. Therefore, we'll cover the same steps more than once.

We're showing the full paths for **Xapollo** and **xclock** here, but in the future we'll expect you to know the paths and/or to have entered **/usr/bin/X11** in your path environment variable (a link is installed to point from there to **/etc**):

```
/etc/Xapollo &
/usr/bin/X11/xclock -display :0 &
ps -ax
kill xclock's_PID
kill Xapollo's_PID
```

The ampersand ("&") runs the process in the background, so your shell isn't held captive. The **xclock** provides visual proof that X is running: if you move the cursor over the **xclock**, the cursor will change into an "X." The **ps** command lists the process IDs of all your processes. You can't stop **Xapollo** or **xclock** without knowing their process IDs.

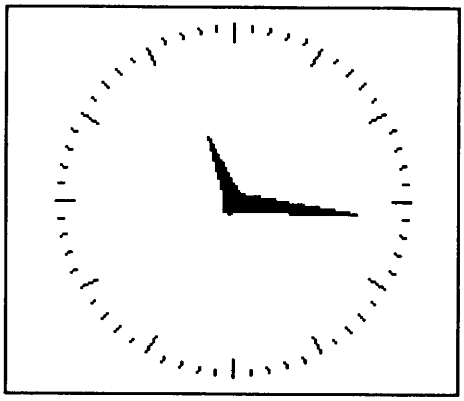


Figure 1-1. The **xclock** *Client*

If you started **Xapollo** "by hand," it won't go away when you log out (unless you used **startx**), but after a few minutes of inactivity, you'd be asked if you wanted to "blast" the processes: "Unable to stop all processes. Want to blast them? (Y/N)." *Never blast a process* because paging files and other operating system information will not be cleaned up properly until you reboot your workstation. If you must blast a process, be sure to reboot your workstation immediately.

The server, despite its name, is not known to the DM as a server—as you can check with the **/com/pst -ty** command. This means that the X server will not automatically continue to run after you log out (if you start it while you're logged in), and the DM will not be able to stop the server at logout. You'll have to issue the command

kill *Xapollo's_PID*

This will stop your X server and any clients.

1.2 Basic Components of Domain/X11

Domain/X11 has several important parts:

- **Xapollo**, Apollo's implementation of the X server that runs on Apollo systems with the existing Apollo window system.

- The Xlib–C library, the programmer's interface to the X Window System. Apollo has added functionality to Xlib to make it possible to run Apollo native graphics programs (GPR, GSR, 3D GMR, PHIGS, etc.) in an X window.
- Supported X clients: **xterm** (a terminal emulator) and **uwm**, a popular X window manager.
- Items from the MIT release tape: MIT documentation and man pages for various X clients.

Domain/X11 relies on the appropriate UNIX environment of the Domain operating system. Domain/X11 installs on individual nodes as local copies or links, in a standard procedure using Apollo's Release and Integration Tools.

Refer to the Apollo release document(s) in **/install/doc/apollo** for specifics on the appropriate version of the Domain operating system and for

- Details on Apollo's support for the various components of Domain/X11
- A list of files, related documentation, and limitations

See also the section "Does This Manual Support Your Software" in the preface.

1.3 Introduction to the X Window System

The X Window System (or "X") was developed at MIT beginning in 1984 under the aegis of Project Athena and the Laboratory for Computer Science. X is developed and maintained by a consortium of industry leaders who have provided direction and have contributed to its development and funding. It has been adopted by many computer manufacturers because it is widely portable, and applications developed using the Xlib calls can be re–compiled and run on other systems that have the same version of X.

X provides a common, network–transparent, graphics–oriented window system. **Windows** are the rectangular areas on your **screen**, the physical, TV–like device commonly called a monitor. The windows can overlap each other, in which case they're in a **stack**. They can be used to run programs (processes). Their placement, size, and order (which one is on top) are controlled by **window managers**, which are programs that let your display screen keep track of one or more windows whose sizes and locations you can change by menu options or by point-and-click methods (using a mouse, for example).

All windows can contain other windows. They are designed to be inexpensive resources: applications using several hundred windows are possible (though uncommon). Windows are often used to implement individual user interface components such as scroll bars, menus, and command buttons.

The X Window System can run multiple applications simultaneously in multiple windows, generating text and graphics in monochrome or color on bitmap displays. X provides for generating multifont text and two-dimensional graphics (such as points, lines, arcs, and polygons) in a hierarchy of rectangular windows.

1.3.1 The Client/Server Model

X follows a network "client/server" model. An X **client** is an application program that uses Xlib or Xt Toolkit calls. (See Figure 1-2.)

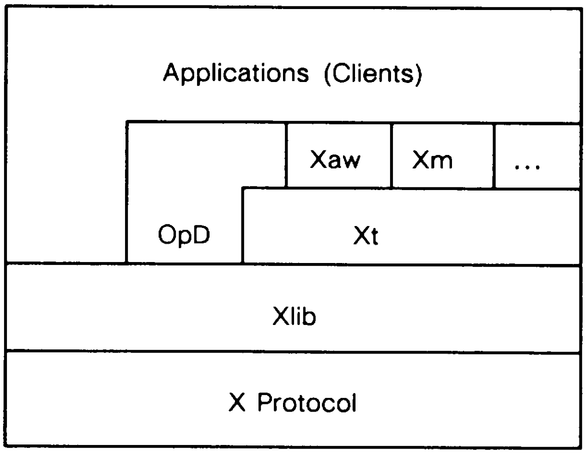


Figure 1-2. Layers of X Libraries

Xlib is a lower-level library that provides a procedural, C language interface to the X protocol. The Xt Toolkit is the layer above Xlib; it makes it easier to write a library of **widgets**, which are typically user interface components such as text labels, scroll bars, command buttons, and menus.

Most X clients combine Xlib and X Toolkit calls. The "Xaw" in Figure 1-2 is the Athena widget set; "Xm" is the OSF/Motif widget set; "OpD" is Open Dialogue. See Section 1.6.1 for more information about libraries and widget sets.

Clients can be programs such as databases, text editors, graphics programs, and window managers. They can each run on different computers on the network, and they can display their output on one or more screens, taking input from keyboards and mice associated with those screens. Typically, a client can display its window or windows on any workstation in a network that is running an X server that will accept a connection to the client.

The X **server** is a program that interacts with the graphics display, input devices, and client software to accept all input from the user —and from any client—and update the display as appropriate. User input is typically from the keyboard and mouse. The server thus provides display services for clients. (Apollo's server is named **Xapollo**.) A single server can provide display and input services for multiple clients, and a single client can use multiple servers.

Since the client and server communicate through a device-independent protocol, they can be running on entirely different system and hardware types. This means that you can run a program on the system that best suits it, while viewing the results on whatever type of workstation you happen to have.

1.3.2 Client/Server Communication

The client and server can be on the same or separate machines. If they're on the same machine, they can use UDS (UNIX Domain Sockets) interprocess communication for their "link." If they're on separate machines, they must use TCP.

In UNIX, the pipe is an I/O mechanism with two ends, or **sockets**. Data written into one socket can be read from another. Sockets created by different programs use names to refer to one another. The names are translated into addresses that come from an address space called a **domain**. When your clients don't use TCP to communicate with their server, they use UDS interprocess communication.

We strongly recommend that you use TCP rather than UDS communications for the following reasons.

- If by chance you start *two* X servers with UDS connections, there will be confusion about which client is connected to which server.
- It takes longer to start the X server without TCP, and you may not know whether it's working.
- You may want to run a client on a different workstation, but you won't be able to if you don't use TCP.

The X server attempts to create a TCP connection when it first comes up. If this does *not* succeed, it will continue the attempt for approximately one minute, and then it will create a UDS connection. If both connections fail, the server will exit.

TCP/IP Communication

An Apollo server with clients on different systems has to communicate through a TCP/IP link. TCP/IP (Transmission Control Protocol/ Internet Protocol) is a standard protocol, originally developed by the Defense Advanced Research Projects Agency (DARPA). It works on various types of computers so users can share resources among many different machines.

Computers that communicate with one another via TCP/IP communications are called **hosts.** A **gateway** node contains the appropriate software and hardware to provide the physical link between different networks. When one network is connected to other DARPA-conforming networks, a **TCP/IP internet** is created. A **Domain internet**, on the other hand, contains networks that are *all* running the Domain distributed environment: these networks use communications protocols such as the Apollo Token Ring (similar to IEEE 802.5) and the IEEE 802.3, commonly called ETHERNET.

Figure 1–3 illustrates a simple TCP/IP implementation in a Domain internet that includes two Apollo Token Ring (ATR) networks and one IEEE 802.3. The Apollo workstations communicate with each other using Domain protocols, and with the HP/UX workstation using TCP/IP. (For more information about TCP/IP on Apollo systems, see *Configuring and Managing TCP/IP*.)

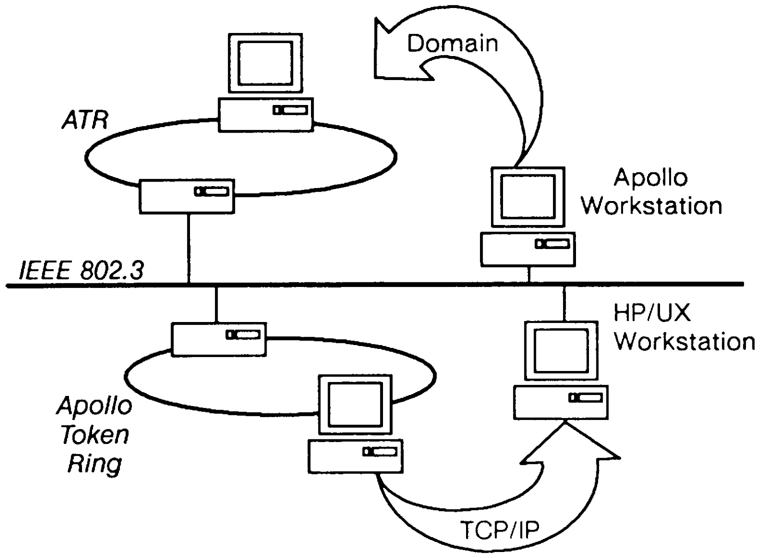


Figure 1–3. Domain Internet with Protocols

TCP is an integral part of your workstation environment, but a system administrator will have to assign you an **internet address** in the file **/etc/hosts**. Afterwards, you may receive a typical email message like the following:

```
Your node //apollo has been added as a TCP/IP
host. The internet address is: 205.7.03.44
```

The most common applications using TCP/IP are the Network File System (NFS), remote login, and file transfer. Admittedly, this doesn't sound like much if you're used to the capabilities of networked Apollo workstations. However, on some other networks, you'd have to use the **rlogin** command on your workstation to "remote login" to the workstation whose file you wanted to see, set the DISPLAY variable there to direct the display back to your own workstation, run an **xterm** client, and finally view the file.

1.4 Your Workstation and X

Figure 1–4 shows the interrelationships of the major components of the X Window System. This figure shows an X client running on a Series 10000 but using your DN3500 workstation for the display. The X client contains application code that makes calls on Xlib, which translates them into X protocol requests to the server running on the DN3500. These requests are transmitted by means of the TCP/IP protocol.

The client could be running on other HP workstations, or on any other system that supports Version 11 of the X protocol and uses a TCP/IP connection to the server. The client can be written in portable code which must be compiled on each unique host system and linked with its Xlib.

The device-independent layer, **dix**, shown as a "request dispatcher," lets the server communicate with any client, regardless of what type of machine it's running on, as long as both use the same version of X. The "translator" layer (**ddx**) in Figure 1–4 translates requests by the client—for instance, to draw a circle on the display—into GPR calls that its host machine (the DN3500) actually understands. Thus when the client's code uses the Xlib **XDrawArc** call to ask the server to draw a circle, an appropriate Apollo graphics routine (perhaps **gpr_$circle**) is actually used.

The **dix** layer of the server is basically portable. All the X servers for the same version of X have much of this layer in common.

The other layers contain the code that Apollo has modified and included from Apollo-specific libraries.

It's important to realize that *the client specifies the display*. This means that the client specifies which server to use. (The server is run without specifying a display, because it listens for new connections over both TCP and UDS).

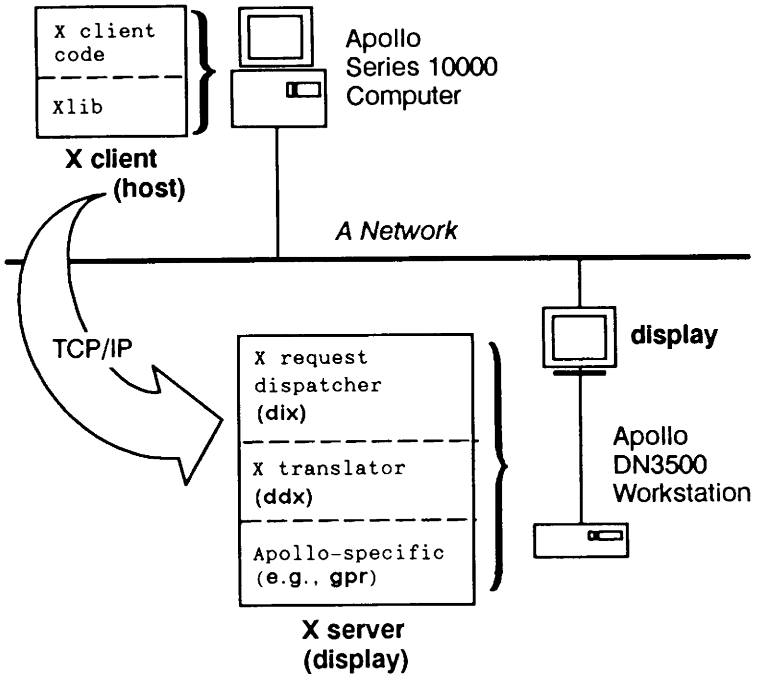

X client code
Xlib
X client (host)
Apollo Series 10000 Computer
A Network
TCP/IP
X request dispatcher (dix)
X translator (ddx)
Apollo-specific (e.g., gpr)
X server (display)
display
Apollo DN3500 Workstation

Figure 1-4. Client, TCP/IP, Server, and Display

1.5 The Shared Server Implementation

Various users require a different mix of X and DM functionality that includes applications engineered for a particular window system. Naturally, some users prefer to work in a DM environment, but they also wish to run software developed for the X Window System. Other users want to work in an X environment, but still be able to run DM-based software.

To allow these varied uses, **Xapollo** provides **DM-root** and **X-root** modes of operation, depending on which window system owns the root (background) window. The **Xapollo** server lets a workstation operate in one and only one of these two modes at any given time.

1.5.1 Definition of Share Modes

Xapollo is called a **shared server** because it allows X and DM windows to be displayed on the same screen at the same time. The two share modes can be distinguished as follows:

- When the DM owns the root, it acts as the primary window system: it "owns" the background window, draws the cursor as an arrow, and does all window management (move, resize, create an icon, etc.) for both X and DM windows.
- When X owns the root, X is the primary window system: it "owns" the background window and draws the cursor as an "X." DM window management has been explicitly turned off. An X window manager such as **uwm** does all window management and moves, resizes, and makes icons out of both X and DM windows.

"Shared server" does not mean that one program has been written that acts as both an X and a DM window system. X and the DM remain distinct systems.

In either mode, both DM and X windows can share the screen. Thus, DM pad calls and Apollo native, graphics-based programs do not have to be recoded, recompiled, or relinked to work with **Xapollo**. This makes it possible, for example, to use the Apollo debug utility (**dde**, which uses DM pad calls) to debug an X client while you view both the client and the debug utility on the screen simultaneously.

1.5.2 System Foundations for Share Mode

X and the DM are able to coexist because of a pair of low-level Apollo system facilities that are called the Rectangle Manager (RM) and Input Demultiplexor (IDM). These utilities together form a window system in the abstract. Normally, they are invisible to the user.

The RM manages a workstation-global database of rectangular areas on the screen. The database contains information on size, location, the hierarchical stacking order of the rectangles, parent/sibling relationships, obscured windows, and window ownership by processes. The RM does not "draw" anything on the screen—the owning process does.

The IDM controls workstation input from the keyboard and mouse. It keeps a database of hierarchical input focus trees (related to the RM's rectangles) and event queues, and it provides a mechanism to bind particular devices, events, and processes together.

Both the Domain/X11 X server and the DM use RM calls to create and manipulate windows. Both use IDM calls to handle input events. Because both share the RM and IDM databases, they can manipulate each other's windows and properly obtain input events. This allows, for example, the keyboard focus to pass back and forth between X and the DM as the cursor slides over a mixture of X and DM windows on the screen.

1.5.3 Borrow Mode Servers

The **Xapollo** server is different from **borrow mode** servers. The ADUS (Apollo Domain Users' Society) Apollo X11 and MIT–tape X11 Apollo servers are both borrow mode servers, which means that the X server takes over the entire display. You can switch back and forth between X and the DM, but you cannot display both types of windows at once.

1.6 X Clients, Objects, and Libraries

X clients, as we've said, are application programs that use Xlib in one form or another. If they use Xlib directly, and not any library built on top of Xlib, then they're more monolithic than clients that use various higher–level libraries. On the other hand, the clients that use the various toolkit libraries are **object–oriented**. That is, they typically use the data and functions belonging to the objects that have been built from the toolkits, without having to "understand" exactly how each object is implemented. This allows the objects to be "re–used" in other contexts.

1.6.1 Common X Objects and Libraries

It's easy to be confused by the various X libraries and the object sets made from them. This section briefly describes the difference between some of the X libraries, X Intrinsics, HP widgets, Athena widgets, and Motif widgets.

Xawlib	The library for Athena widgets (see below; this is supplied).
Xlib	The C language interface to the X protocol. Xlib includes the standard, core graphics routines in X (this is supplied).
Xmlib	The library for OSF/Motif widgets (see below; this is not supplied).
Xrlib	HP's old X10 library (not supplied).
Xtlib	The standard, base X Intrinsics library for building user interface toolkits. In all cases, you cannot expect binary compatibility from release to release, but you may expect source compatibility.
X Intrinsics	Same as Xtlib.
Athena Widgets	Available on the MIT X tape, but not an official part of the Domain/X11 release.
HP Widgets	A version of HP's toolkit (based on the Intrinsics) on the MIT X "Contrib" tape. Some of the HP widgets are also available in the OSF/Motif toolkit, but most have been changed significantly. They require a modified version of the R2 Intrinsics also available on the contrib tape with them, but not supplied as part of Domain/X11.
OSF/Motif Widgets	To be available from OSF, and from Apollo via Open Dialogue version 2.0, They require a modified version of the Release 3 X Intrinsics to run.

1.6.2 Popular X Clients

The X Window System distribution from MIT comes with a number of X clients (programs). The binaries are in **/usr/bin/X11**, and each has an associated **man** page. They include supported and unsupported clients.

Supported Clients

xterm	A terminal emulator, **xterm** lets your Apollo workstation act like a VT102 terminal, which can be useful when you want to run other X clients or if you log onto multi-user systems over the phone line. Since X lets you run more than one client at a time, you can run several **xterms**.
uwm	The Universal Window Manager. It arranges the windows on a screen and controls the user interface for selecting the window to receive input.

Unsupported Clients

bitmap	A program for creating and drawing bitmaps.
xbiff	A mailbox flag.
xcalc	A pocket-style calculator (Figure 1-6). It also has an **-analog** switch that produces a slide rule with digital readout.
xeyes	Faceless eyes (Figure 1-5) that look around the screen as you move the X cursor in X territory.

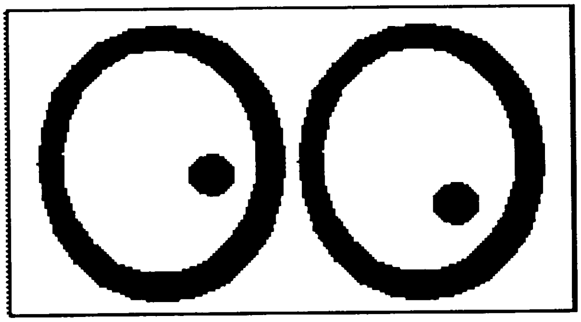


Figure 1-5. The xeyes *Client*

xclock	An analog or digital clock (see Figure 1-1).
xedit	A simple text editor with a small set of Emacs-like commands.
xev	A utility that displays events for a window.

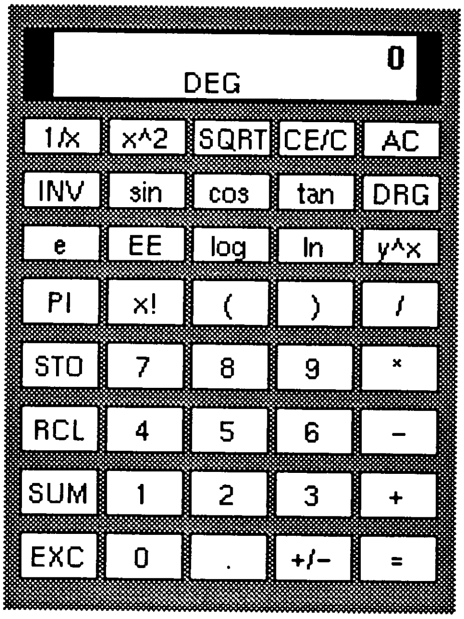


Figure 1-6. The **xcalc** *Client*

xfd	A program that displays X fonts.
xhost	A program that controls the access of other "host" systems to your workstation's X server.
xlsfonts	A program that lists all of the X fonts available.
xmh	An interface to the Berkeley **mh** mail system. (You must supply the **mh** back end yourself.)
xrefresh	A program that refreshes all windows on the screen, including both X and DM windows.

xset A program to set user preference options such as font path and autorepeat, typically for the display and keyboard.

xsetroot A program that customizes the background of your display when X owns the root. (See Figure 1–7.)

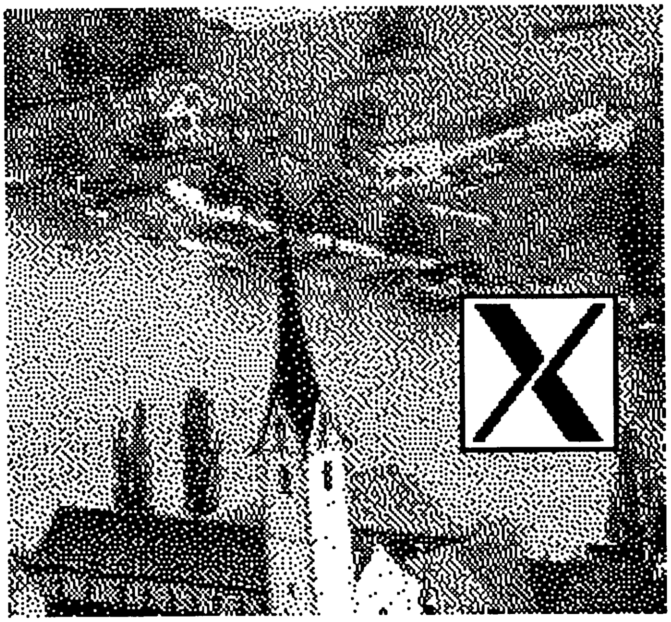


Figure 1–7. **xsetroot –bitmap //zz/images/xbm/rhine.xbm**

xwd, xpr, and **xwud**

The **xwd** client creates file images of X and DM windows that can then be formatted for PostScript printing with **xpr**. The images can be "undumped" back to the screen with **xwud**. (Positioning the cursor in the root window dumps the entire screen.)

xwininfo A program that prints location and size information on the window indicated.

Chapter 2

Starting X and Using uwm

Contents

Chapter 2

Starting X and Using **uwm**

To use the X Window System on an Apollo workstation, you need to know how to start it and how to use an appropriate window manager. This chapter explains how to

- Start the server at boot, login, or run time (*and why you shouldn't start it at run time*).
- Specify root ownership and various configuration files that should be run when you run X.
- Run **xinit** and recognize the use of its associated server and client startup script files.
- Find and look at the server's error log file.
- Run the **uwm** window manager.

After you have read this chapter, you will have a good understanding of X's configuration requirements, including the important concept of root ownership. You will also know how to start the X server, how to use the DM or **uwm** window managers, and find the error log.

You may wish to refer to the Checklist in Appendix A for an overview of the major steps in starting and using X.

2.1 Before Running Xapollo

Before you run the **Xapollo** server or any of the clients, check that you

- Are running the appropriate version of the Domain operating system and have properly installed Domain/X11 as part of the base software (see the release document and this manual's preface if you have any doubts).

- Have at least 16 "pseudo-ttys" in /**dev**. Without enough, you'll get an error message, "Insufficient Pseudo Terminals," when you try to run **xterm**. You can tell how many you have by using the command

  ```
  ls /dev/*typ*
  ```

  If the installation process did not create these, ask your system or network administrator to issue the following commands. (The **su** command is usually privileged—if you can't use **su** because you're not a member of the BSD4.3 group "wheel," use the **login root** command.)

  ```
  su root
  /etc/mkdev /dev pty
  ```

  The **mkdev** command creates 16 pseudo-ttys.

- Have an updated /**etc/sys.conf** file that lists the library or libraries that contain the symbols the clients will resolve at run time. This allows the clients to be much smaller. (This file should be properly updated when the software is installed.) See Section 5.2.2 for more information.

- Have rebooted your node after completing the installation and updating your /**etc/sys.conf** file. (Logout with the **shut** command, and then reboot by issuing the boot PROM's Mnemonic Debugger **ex domain_os** command.)

Often files in /**etc** are soft-linked to '**node_data/etc**: for example, /**etc/rc** is a link to '**node_data/etc/rc**. This is done to allow a diskless node to have its own copy.

2.2 Running the Xapollo Server Automatically

This section describes the steps you need to take to

- Set up X to run as a system server at boot time.
- Start X automatically at individual login.
- Run X in borrow mode.

This section explains how to start the X Window System automatically as a background system server at workstation boot time, so that it is available to anyone who logs in to the workstation. To make running the X server an individual decision and to rely upon each person to start the server as part of the login process (or to run it as needed after login), see Section 2.2.4.

You may confront problems related to "serial re-usability." Perhaps the original X server ran in a DM-owned root, and the next person wants to use the workstation with X owning the root. Or perhaps the default is X root ownership with the window manager turned off, and somebody who has never heard of X doesn't understand why DM window management isn't available. These are very real problems for somebody who must configure a "public" workstation to be used by a succession of different people.

2.2.1 Workstation Initialization

You should know what happens when your workstation boots, so you understand what files are involved and how they relate to starting the server at boot time.

The workstation initialization sequence runs the program **/sys/env**, which loads the symbols from the run-time libraries using the information in the **/etc/sys.conf** file, and then loads and runs the **/etc/init** process as Process 1.

The **init** process reads the file **/etc/rc** (which is a link to **'node_data/etc/rc**) and invokes any processes specified there; it then reads the **/etc/ttys** file and invokes **/etc/dm_or_spm**, which initializes the serial lines and starts the DM (Display Manager), if specified. Once the DM starts, the boot startup files in **'node_data** are executed.

2.2.2 Starting the Server in a DM-owned Root

The easiest way to start the X server at boot time is to do so from your **/etc/rc** file, which runs as root. In DM-owned roots that share the screen with X but use the DM for window management,

```
        touch /etc/daemons/Xapollo
```

and check the end of **/etc/rc** for something like the following lines:

```
if [ -f /etc/Xapollo -a -f /etc/daemons/Xapollo ];
           then
           (echo  "Xapollo\c" >/dev/console)
           /etc/Xapollo -K \
                   /usr/lib/X11/keyboard/keyboard.config -D1 s+r- &
fi
```

The lines in **/etc/rc**, whatever they are, should be just what you need. (The **-f** flag—type **man 1 test**—asks if the file exists. See Section 2.3.2 for information about the switches and keyboard configuration files associated with starting **Xapollo** in a DM-owned root.)

You can use an optional **nice** command in your file:

```
        nice --10 /etc/Xapollo . . .
```

This sets the scheduling priority for **Xapollo**. We recommend not using a higher priority than -10. (The DM runs at -20, which is a higher priority than -10.) Note that you have to be logged in as root if you want to issue **nice** on the command line with these negative priority values. You also have to be root to edit the **/etc/rc** file.

2.2.3 Starting the Server in an X-owned Root

It's easier to start X in a DM-owned root, but it can also be started automatically in an X-owned root if you follow these steps.

1. Become root and change the line in **/etc/rc** (see Section 2.2.2) that runs **Xapollo**, so that the keyboard configuration is not used and so that **r+** specifies X root ownership:

```
        /etc/Xapollo -D1 s+r+ &
```

2. Set the default DM window management **off** by uncommenting in '**node_data/startup.***xxx* the following line, which is typically *after* all the other DM window manager calls (e.g., after **inv -on**):

```
        wmgr -off
```

   With this line, the DM reverts to **wmgr -off** (i.e., to *not* owning the root) at logout, even if a user had been using the DM and a DM-owned root during the logged-in session. (The DM functions disabled by **wmgr -off** are listed in Table 2-1 in Section 2.3.3.)

3. Create the empty file **/etc/daemons/Xapollo**:

```
            touch /etc/daemons/Xapollo
```

   This will cause the X server to be started the next time your node is rebooted. (Reboot your node now to start the X server.)

4. Start an **xterm** on user login. Otherwise, when a user logs in, a DM pad is created because of the script in **/sys/dm/startup_login.***xxx*. This file can be changed to start an **xterm** instead of a DM pad (*only* if the X server is already running). Replace the line that starts the DM pad with

```
cpo /usr/bin/X11/xterm '-e' '/sys/dm/login_sh' '-d' ':0'
```

   This starts an **xterm** with a UDS connection (**-d :0**) running a login shell (**-e /sys/dm/login_sh**).

   NOTE: All other X clients should be started by individual users as part of their login mechanism.

5. When editing **/etc/rc**, please don't attempt to use **xdm** as it is support for a future product.

2.2.4 Starting X Automatically at Individual Login

To start X at login, edit your C shell **.login** file or your Bourne or Korn shell **.profile** file to include the line

```
/usr/bin/X11/startx
```

Your personal **.login** and **.profile** files must list you as the owner: an owner listing of "none" doesn't work. If they won't execute, consult your system administrator.

The **startx** script is just a sample implementation of a slightly less primitive interface than **xinit**. It looks for the user **.xinitrc** and **.xserverrc** files, then system **xinitrc** and **xserverrc** files; if it doesn't find them, it lets **xinit** choose its default. (See Section 2.3.4.) The system files are in **/usr/lib/X11/xinit**; see the **.Xmodmap** resource file there, as well. (Actually, this **xinit** directory is a link to **'node_data/etc/xinit**.)

Your site administrators may have modified **startx** to be more appropriate for your environment.

2.2.5 Running X in Borrow Mode

You can also run **Xapollo** in borrow mode by following these steps:

1. Make your own copy of the sample **/usr/lib/X11/keyboard/keyboard.config** file and define a **Quit** key so that you can get out of borrow mode and back to the DM. (**Quit** is essential if you have no X windows on the screen.) Note in **keyboard.config** that we've defined the **Quit** key to be **F9**. No matter what you define it to be, you also have to press **Ctrl–Shift** at the same time as your Quit key.

2. Use the **xinit** client to start both the server and an **xterm** (the default client) by issuing the following command (there are two literal hyphens after **xinit**, and there is no space between them):

   xinit -- /etc/Xapollo -K *your_kbd.config* -D1 s-r+

   See Section 2.3.2 for more about this keyboard file.

2.3 Starting Xapollo at Run Time

Generally, you'll want to start the server at boot or login time. However, we'll tell you how to start it "by hand" so you can learn what's happening more easily. If you continue to want to start it at run time, use the **startx** script discussed in Sections 2.2.4 and 2.3.4.

You can type

```
man Xapollo
```

for a synopsis of Apollo-specific server options and features, or you can type

```
man X
```

or

```
man Xserver
```

for more general information. The examples in this section work in both C and Bourne shells. The **xmodmap** client for specifying keyboard configuration is common to both DM and X root-owning modes; it's described briefly in Section 3.3.5 and more fully in 4.2.

2.3.1 Quick Hints

Issue the command

```
startx  &
```

then wait a little bit until your cursor returns (you may have to press the RETURN key a second time), and you will see the **xclock** in one corner and an **xterm** in another (provided your system administrator hasn't changed the original **startx** script).

If you want to run more clients "manually" at this point, remember to **kill** them and the server before you log out. (Otherwise, you may be asked if you want to "blast" them, which wouldn't allow the system to clean up after them.)

Your Environment Variables

You should set your environment variables (probably in your **.cshrc** or **.kshrc** files) so that your DISPLAY is *hostname*:**0** and your PATH includes **/usr/bin/X11**, as described in Section 3.1.2. (If you don't set DISPLAY to *hostname*:**0**, your clients will automatically use UDS communications with the server according to the **setenv** command in **'node_data/startup.***xxx*.)

Clients and Configuration Files

Depending on root ownership, various optional startup files should be available and/or run when the X server starts up (see Figure 2-1).

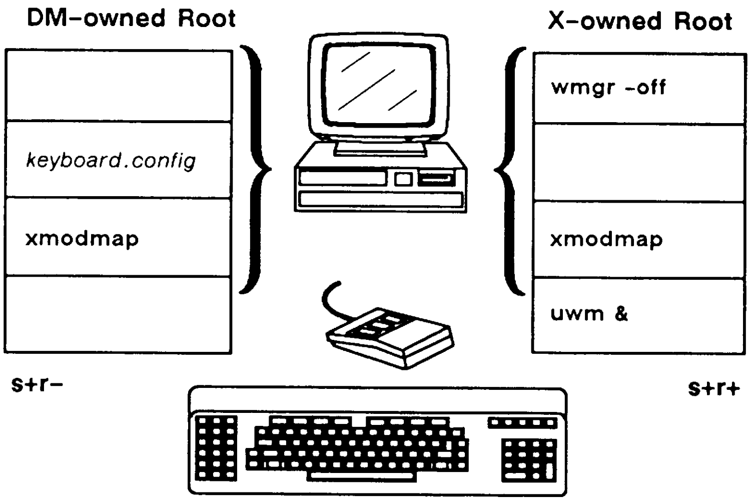


Figure 2-1. X Server Startup

Xapollo starts best (in DM-owned roots) when a keyboard configuration file is specified and when **xmodmap** is run. The file reserves certain keys for use by the DM, and other keys for use by X—if you don't use it, all the keys will be passed directly to the X server, including those you may want the DM to handle, such as POP or MOVE.

When X owns the root, the keyboard configuration file is not wanted, but the DM command **wmgr -off** and the X client **uwm** need to be used to turn window management over to X.

2.3.2 Starting in a DM-owned Root

To run the X server so that it shares the screen but lets the DM still own the background and handle window management, issue this sequence of commands (use the file **.kb2sample** for keyboard 2—if you're not sure what keyboard you have, check the Glossary):

```
Xapollo -K /usr/lib/X11/keyboard/keyboard.config \
                                          -D1 s+r- &
 .
 . (pause as the X server initializes)
 .
xclock &
xmodmap /usr/lib/X11/keyboard/xmodmap.kb3sample
```

(We put the **Xapollo** command on two lines to fit it on the page, but you should use only one line, without the backslash. You may use the exact files we've named here, but we've put their names in italics to help indicate that you can use *any* appropriate file of your own.)

The following paragraphs explain the switches used on the **Xapollo** command; for a complete description of all available switches, refer to the **man Xapollo** page.

-K */usr/lib/X11/keyboard/keyboard.config*

Use the named keyboard configuration file. (See Section 4.1 for a complete discussion of this file.)

When the DM owns the root and is acting as the window manager, *this keyboard configuration file is necessary to use DM window management functions* to move, grow, and pop X as well as DM windows. The sample keyboard configuration file specifies that X will get all the white keys, F0–F9, the arrow keys, the keypad on keyboard 3, and the five upper right keys when they are typed in X "territory." The DM will get the keys that really have meaning only to the DM (especially the DM window management keys, such as MOVE/GROW, NEXT/WNDW, and POP), even when they are typed inside an X client.

-D1 s+r-

-D1 A flag that specifies Apollo-specific options for a display.

s+r- Options on the D flag to specify display sharing and root ownership. The default shown here specifies: share the display with the DM (**s+**), do not own the root (**r-**).

& Run the server in the background, where **Xapollo** almost always runs.

The X server can take up to 60 seconds or longer to initialize fully. The length of time is related to the number of hosts listed in the **/etc/X0.hosts** file for which the server must retrieve real network addresses. It also varies depending upon CPU speed, network traffic, system load, TCP, and UDS connections.

See Section 4.2 for a discussion of the **xmodmap** file used in this sequence of commands. (The file gives you modifier keys that work with the mouse.) See Section 2.3.4 for a discussion of why we're using the **xclock** client in this sequence: essentially, it keeps the modifier keys issued by the **xmodmap** file from being reset. If you use **startx**, you don't have to worry about the server resetting.

2.3.3 Starting in an X-owned Root

To run the X server so that it shares the screen with the DM but takes over window management, issue this sequence of commands (use **.kb2sample** for keyboard 2):

```
/usr/apollo/bin/xdmc wmgr -off
Xapollo  -D1 s+r+ &
  .
  . (pause as X server initializes—cursor turns into an "X" if
    over no window)
  .

xclock &
xmodmap /usr/lib/X11/keyboard/xmodmap.kb3sample

uwm -f /usr/lib/X11/uwm/menu.uwmrc &
```

Table 2-1. The DM Functions Disabled by **wmgr -off**

Function	**Purpose**
wm	Move a window across a screen.
wme	Move a window across a screen with rubberbanding to outline its progress.
wg	Grow or shrink a window.
wge	Grow or shrink a window with rubber-banding to outline its progress.
wi	Make a window invisible.
icon	Make a window into an icon or vice versa.
idf	Set icon positioning and offset.
wsel	Select a window by making it a full window (no icon or invisible).
wp	Push or pop a window on the stack.
rs	Refresh the screen.
inv	Invert monochrome screen to black on white, or vice versa.
bgc	Turn on or off the gray-scale back-ground color (monochrome monitor).
mono	Set color monitor to black and white.

The significant changes to the **Xapollo** command discussed in Section 2.3.2 are that X owns the root because of the **r+** switch, and there is no keyboard configuration file. There is no benefit from excluding keys with a –K *keyboard.config* option when X owns the root, because the DM won't receive the key events. You could modify the sample file in **/usr/lib/X11/keyboard/keyboard.config** to create a server **Quit** key that would exit the X server whenever pressed in case of a server problem, but this is mostly intended for borrow mode—see Section 2.2.5.

The **wmgr** command is a DM command that turns off DM window management and DM root ownership so that X and the DM do not conflict, possibly locking up the display. As part of disabling DM window management, all DM icons are changed into windows, and all windows that the DM had made invisible are made visible. Table 2–1 lists functions disabled by the **wmgr -off** command.

Whenever you shut down the root-owning X server, you can restore the DM's ability to manage windows by pressing <CMD>, then typing **wmgr -on** in the command area.

The DM command **dmio** can be used for additional control:

- **dmio -on** puts up DM windows permanently.
- **dmio -off** only puts them up when needed, and then only until input is complete. (For example, **dmio -off** will put up the Command window when you press the CMD key, and then remove it after you press RETURN.)

The **xmodmap** file is discussed in Section 4.2.

Lastly, start the **uwm** X window manager as explained more fully in Section 2.5.2.

2.3.4 xinit and the Initial Client

We suggest you use the **xinit** program for starting the server at run time. The **startx** script we provide uses **xinit** in a recommended way.

When you run the X Window System in share mode with the DM or X as the primary windowing system, at least one client needs to be run *before* any **xmodmap** or other resource command. (See Section 2.2.4 for a sample startup script.) This initial client doesn't need to do anything, but, like any client, it will add to the connection count kept by the server.

Server Reset

When this connection count becomes zero, the server **resets**. This means that the server reinitializes itself, clears all state information (such as keyboard definitions), and frees all resources (pixmaps,

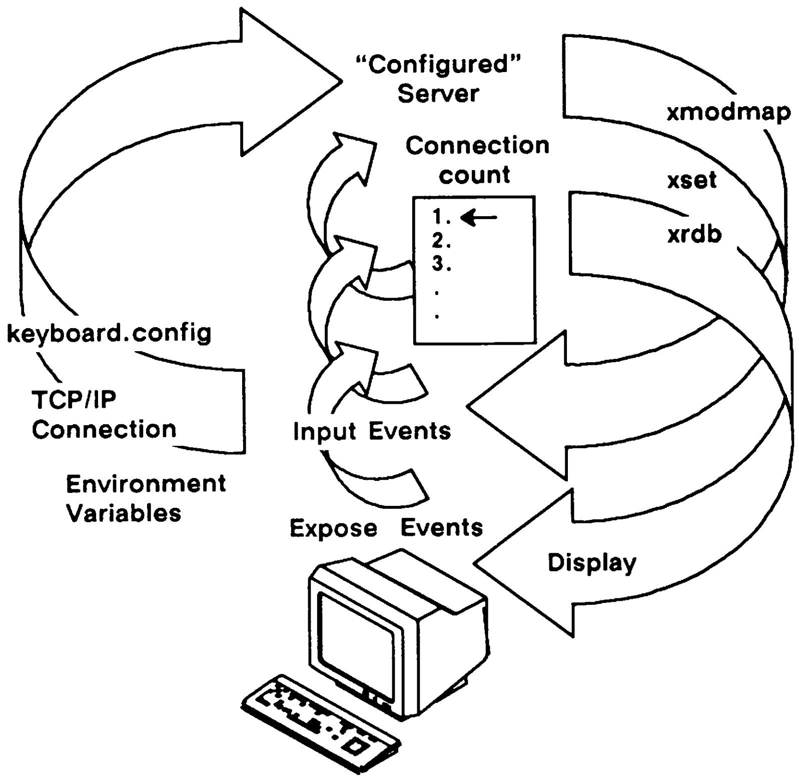


Figure 2–2. Server Startup and Reset

fonts, graphics contexts, etc.) that clients have asked it to save. Typically, these various resource definitions have been created by **xmodmap**, **xset**, or **xrdb** commands. These resource-oriented clients, however, do their jobs and then quit, so they're no longer included in the connection count. Figure 2–2 shows that the server in a sense juggles the resource definitions as long as the connection count remains above zero.

If you've just typed the following commands in a DM pad to run the server in share mode

```
Xapollo &
xmodmap  /usr/lib/X11/keyboard/xmodmap.kb2sample
```

the **xmodmap** client redefines some of the keys, but the redefinition has absolutely no effect, since the server (in this example) resets immediately after the **xmodmap**, because the connection count becomes zero as soon as the **xmodmap** finishes.

Resetting in this manner is a normal procedure for an X server that is a system resource available to many users who log in and out one after the other. When the server resets at logout (because the connection count has become zero), none of the customization or "configuration" remains to confuse the next user, who is responsible for "reconfiguring" the server as appropriate. In the Apollo shared environment, the number of X clients could go to zero while you were still logged in and had DM windows on the screen but no X windows. The server would then reset itself, destroying state information that you might want to save while you were logged in.

The xinit Program

The **xinit** program first runs the server and then a special, "initial" client. Then, **xinit** keeps the connection to the server open as long as the special client is running; when the special client goes away, **xinit** closes the connection and stops the server. This is why we use **xinit** in our **/usr/bin/X11/startx** script .The simplest syntax for **xinit** that specifies both the client and server is

xinit *client_script* -- *server_script*

However, the **xinit** program can be run without any arguments, in which case it starts **Xapollo** and then **xterm**. If the arguments are specified,

- It looks for the *server_script* file (named, for example, **.xserverrc**) in the user's home directory and runs it as a shell script to start up the server. If no such file exists, it issues the command **X** as a default.

- It looks for the *client_script* file named **.xinitrc**, for example, in the user's home directory and runs that as a shell script to start up client programs. If no such file exists, it starts an **xterm** using a UDS connection.

- The script names *must begin* with a slash or period.

The programs run by the scripts should be run in the background (with &) if they don't exit right away, so that they don't prevent other programs from starting up. The client script can run any number of clients in the background, but *must* **exec** *the last client in the foreground*. Typically, this special client is an **xterm**; when the user exits it, the server terminates. The server script generally just **execs** the X server in a desired root mode.

The sample **startx** script can be run with the same arguments as **xinit**, i.e., with a client and a server script. It also can be run without arguments, in which case it uses **.xinitrc** and **.xserverrc** as the default file names, and it looks for them in the home directory first. If it can't find them there, it looks for them in **/usr/lib/X11/xinit**. If it can't find them there, it invokes **xinit** without any arguments.

The DM makes a connection at login and breaks it at logout. (Nothing happens if the server isn't running.) If the server is run at boot time, then this DM connection tends to equate logout with reset.

2.3.5 Sample Startup Scripts

We provide sample client and server startup files for both modes of root ownership in **/usr/lib/X11/xinit**. You can copy these to your home directory.

The Server Scripts

The default **xserverrc** script that we provide merely issues the command

```
exec Xapollo -K /usr/lib/X11/keyboard/keyboard.config \
                  -D1 s+r-
```

which, as we saw in Section 2.3.2, starts the server in a DM-owned root. If you want an X-owned root, substitute **xserverrc.xroot** for the script. You'll notice that we don't use a "." dot at the front of the resource filenames, so they'll show up easily when you do an **ls** on the directory. Copy them to your home directory and put the dot back in front of them, otherwise **startx** won't find them.

The Client Scripts

The client scripts in **/usr/lib/X11/xinit** also come in three varieties, without the leading dots. There's one for DM-owned roots, one for X-owned roots, and the default one, **xinitrc**, for DM-owned roots.

The client scripts include topics that we discuss in later sections, for example

- "geometry specifications" (see Section 3.2.2)
- **xmodmap** (see Section 4.2)
- **xrdb** (Section 3.3.7)
- X resource files (Section 4.4)

However, we'll show you the default **xinitrc** here (see Figure 2-3). By the time you have finished reading this manual, you should understand all of the script, including the **-f** "file exists" switch (type **man 1 test**). For now, though, just keep in mind a few points:

- The important part shows up in the last two lines:

  ```
  xclock -geometry 50x50-1+1 &
  exec xterm -geometry 80x24+0+0 \
                                -name login
  ```

- The **xterm** is the "special client." When it is killed or exited, the server will also go away (and so will any other clients). The special client can be any X client; it just has to be **exec**'d in the foreground. (Of course, this means that it should be on the last line, since any following commands won't be executed.)
- The **xclock** isn't special—you can run as many "non-specials" as you wish, but remember to run them in the background with the ampersand (&).

```
#!/bin/sh

userresources=$HOME/.Xresources
usermodmap=$HOME/.Xmodmap
sysresources=/usr/lib/X11/xinit/.Xresources
sysmodmap=/usr/lib/X11/xinit/.Xmodmap

# merge in defaults and keymaps

if [ -f $sysresources ]; then
    xrdb -merge $sysresources
fi

if [ -f $sysmodmap ]; then
    xmodmap $sysmodmap
fi

if [ -f $userresources ]; then
    xrdb -merge $userresources
fi

if [ -f $usermodmap ]; then
    xmodmap $usermodmap
fi

# start some nice programs

xclock -geometry 50x50-1+1 &
exec xterm -geometry 80x24+0+0 -name login
```

Figure 2–3. The Default **xinitrc** *File*

2.4 Xapollo Error Logging

Xapollo server errors are logged in **/usr/adm/X0msgs** (a link to **'node_data/system_logs/X0msgs**), instead of being sent to **stderr**. The entries are time–stamped and include a traceback if the server crashes. While the server is running, it locks this file, so you cannot open a DM edit pad on it. To view just the 40 latest lines at the end of the file (you must be on the same node as the file), type

```
/usr/ucb/tail  -40  /usr/adm/X0msgs
```

2.5 DM and X Window Managers

This section describes key aspects of two common window managers.

2.5.1 Running with the DM as Window Manager

We expect you to be familiar with the DM window manager or find the appropriate documentation to learn about it. With the DM as the window manager, you can handle X clients and DM programs nearly identically, but

- DM window commands that act on the window or screen as a whole work the same on either DM or X windows, whether issued from the mouse or from keys such as MOVE/GROW, POP, NEXT/WNDW, HELP, or CMD. (This is only true if you specified a keyboard configuration file with a -K option on your **Xapollo** command. If you didn't, none of these DM keys will work on X windows.)

- If you define mouse keys as MOVE, GROW, or POP, however, be aware that X clients (just like normal Domain graphics programs) may intercept mouse events before the DM can act on them. The **xterm** client, for example, traps mouse events, so the DM (which owns the root) never sees them. (You can still initiate move and grow commands by starting with the mouse slightly outside the X window, near a corner.)

- Key commands that somehow affect the contents of the pad displayed inside the window (as with the CUT/COPY key, all editing commands, and the page up and page down functions) work on DM windows but not on X windows. (They are DM transcript pad and frame mode commands.) For example, the CUT/COPY and PASTE keys do not work between X and DM windows. Also, when the DM owns the root, the NEXT/WNDW key cycles among all non-occluded windows (X as well as DM), but you can move the cursor to an X or DM window and issue the DM command **curs -off** so the NEXT key ignores it.

X clients do not take control of existing windows as many DM programs do. If X clients output to the screen at all, they create an entirely new window to run in.

X clients do not use any DM coordinate and size information. They normally start themselves at the default position (the upper left-hand corner of the display) and at a default size. To start them anywhere else, use an X "geometry specification" on the command line, as described in Section 3.2.2.

2.5.2 Running with uwm as Window Manager

The only X window manager that we currently support for Domain/X11 is **uwm**, the Universal Window Manager. Using **uwm** is described thoroughly in Chapter 3 of O'Reilly's *X Window System User's Guide*, Vol. 3.

The **uwm** window manager can use a **.uwmrc** startup file. By default, **uwm** looks for it in your home directory, where you can copy either of the two samples that we provide. The **man uwm** page explains how to customize your own **.uwmrc** file. Also see Chapter 10 in O'Reilly's Vol. 3 for information on customizing **.uwmrc** files.

First, we'll tell you about **uwm** without any startup files, and then step quickly through the **menu.uwmrc** and **mit.uwmrc** files.

You can use the WindowOps menu, a combination of keyboard and mouse, or let **uwm** automatically guide you in sizing and placing a client window on the screen when you run new clients.

Using uwm without a Startup File

Start **uwm** in an X-owned root with the command

```
uwm &
```

When you hear the beep, **uwm** has been initialized and is ready to use.

Menu selection in **uwm** without a startup file is fairly simple. Move the pointer to the root window and hold down the middle button on the mouse. The WindowOps menu will display. Then select the menu option by doing the following:

1. Continue to hold down the middle button.

2. Drag the highlight to your desired item.

3. Release the middle button.

What you do next depends on the menu entry you've chosen. See Table 2-2 for a general synopsis.

If you create a new window, you're actually creating an **xterm** and it may take a few seconds before the "upper-left-corner" cursor appears. Move this new cursor to where you want the new window's upper-left corner to appear, and follow the directions in Table 2-2 (the "New Window" entry) for the window size of your choice.

If you redraw or move a window, move the "pointing hand" (☞) cursor into the window to be redrawn or moved, then click the middle mouse button to redraw, or press and hold the middle button as you move the "cross" to where you want to move the window.

In general, you'll find **uwm** fairly easy to learn. If you need more information, type **man uwm**. Note that the Preferences menu (which you can reach by holding down **Shift-Meta-Button2** and then sliding off the WindowOps menu to either side) has no effect on Apollo systems—see **man xset**.

The menu.uwmrc File for Novices

The file **/usr/lib/X11/uwm/menu.uwmrc** creates a **uwm** that is primarily menu-driven, as often suits a novice user. Menus are produced when you press a mouse button in conjunction with the **Meta** key, which is typically defined by an **xmodmap** command as <F3> on keyboard 2 and <F0> on keyboard 3 (if your **xmodmap** uses the sample keyboard files).

Note that an **Xapollo** that is run at boot time normally doesn't set up any modifier keys. You can run **xmodmap** to do so from your **.login**, **.profile**, or **.cshrc** files, or you can run **xmodmap** and **Xapollo** from the **startx** script.

Table 2-2. Principal **uwm** *WindowOps Menu Entries*

Menu Entry	**Purpose**
New Window	Make a new window (click left mouse for default size, press the middle button and sweep for custom-size, or click right for a maximum-height window).
Redraw and Move	Middle button refreshes a window or moves one with rubberbanding.
Resize	Hold down the middle button and move the window border to the desired size.
Raise and Lower	Left or middle buttons bring a window to the top of a stack or behind all others.
NewIconify	Left button converts a window to an icon or an icon to a window.
Focus	Click middle button to send the typing focus to the chosen window.
Freeze and Unfreeze	Freeze restricts the server's attention to **uwm**, so you can move or resize windows without the time delays when the server talks to other clients.
Exit	Stop **uwm** and cause all icons to revert to windows.

To use the **menu.uwmrc** file, issue the command

```
uwm -f /usr/lib/X11/uwm/menu.uwmrc &
```

Menu selection with this **.uwmrc** file uses a verb-object (or action-window) syntax, whether you're dealing with the "standard" menu or the "extended" menu (see Table 2-3). Press **Meta** and the left mouse button to produce the WindowOps menu, or use the middle button to produce the Extended WindowOps menu. Continue to hold down the mouse button. (Again, if you haven't run **xmodmap**, you don't have any **Meta**.)

Table 2-3. Menu Entries with the **menu.uwmrc** File

WINDOW OPS (Meta plus left mouse button)
(De)Iconify Move Resize Lower Raise
EXTENDED WINDOW OPS (Meta plus middle mouse button)
Create Window Iconify at New Position Focus Keyboard on Window Freeze All Windows Unfreeze All Windows Circulate Windows Up Circulate Windows Down

1. Select the action:

    1. Move down the menu to select the action (or you can cancel by moving off the menu).

    2. Release the mouse button. The cursor becomes a pointing hand, upper-left-corner, or cross, depending on which menu entry you've chosen.

2. Select the window:

    1. Move the cursor to point to a window object.

    2. Press any mouse button.

3. Move the mouse to act on the window.

4. Release the **Meta** key and mouse button.

The mit.uwmrc File for Experts

Another **.uwmrc** file is designed for experienced **uwm** users.

The **/usr/lib/X11/uwm/mit.uwmrc** file creates a **uwm** that is primarily driven by the **Meta** key in conjunction with mouse buttons. Some menus are provided, but they are secondary, as usually suits an expert user. This is a complex file to read and interpret, but see Table 2-4 for a synopsis. ("Mouse in Root" means that the mouse's cursor is in the root window.)

Table 2-4. Button and Key Combinations for Expert **uwm**

Meta with Mouse	**Purpose**
right	Move and raise a window.
center	Resize a window.
left	Change window to icon, or vice versa.

Mouse in Root	**Purpose**
center	Display the menu.

2.5.3 Other Window Managers

Other window managers that you may find available include

- **cwm** (very much like **uwm**) from the University of Michigan's CITI group
- **cwm** (the Andrew window manager called the Cambridge Window Manager) from Carnegie Mellon University

- **hpwm** (the HP window manager)
- **mwm** (the Motif window manager, based on **hpwm**)
- **rtl** (Siemens Research Technology Lab) tiling window manager
- **twm** (Tom LaStrange's Window Manager) "reparenting" manager that draws "window decoration" around clients

Chapter 3

Using X Clients

Contents

Chapter 3

Using X Clients

This chapter explains how to use many of the X clients. Before you can use them on an Apollo workstation, you need to

- Know how to set your environment variables.
- Be aware of some of the common aspects of clients, such as where they display themselves.
- Know how to start most local and remote clients.

We'll also describe briefly a few of the more generally useful clients, and then we'll give you a thumbnail sketch of most of the others. If you need more information about particular clients, see the appropriate **man** page or look in O'Reilly's Vol. 3, *X Window System User's Guide*.

3.1 Setting Your Environment Variables

You need to set your DISPLAY and PATH environment variables so that clients know which device to use for the display, and so that various programs can be found in the correct search path.

A temporary stand-in for the DISPLAY variable may be set on the command line that runs the client, or the DISPLAY variable itself may be set from the command line or in various shell scripts that are run either when you log in or when you start a new shell.

3.1.1 Where to Set Them

C shells (which are created by the **/bin/csh** command) automatically execute a **~/.login** script when you log in. Bourne and Korn shells (which are created by **/bin/sh** and **/bin/ksh**, respectively) automatically execute a **~/.profile** script at login. Type **man chsh** to read how to change your login shell. Basically, enter

```
chsh -s username
```

and respond to the prompt.

Whenever a new C shell is created, it automatically runs a **~/.cshrc** script file, where you can set your environment variables. Bourne and Korn shells, on the other hand, evaluate a variable named either SHENV (Bourne) or ENV (Korn) that may be set in the **~/.profile** file to a pathname that specifies a file that will be executed when these shells start up. That file is where you can set your environment variables for all Bourne and Korn shells. The SHENV or ENV variable may be set by including one of the following pairs of lines in your **~/.profile** file:

```
ENV=pathname
export ENV
```

(or)

```
SHENV=pathname
export SHENV
```

Typically in Apollo documentation, *pathname* is **~/.kshrc** for Korn shells and **~/.shrc** for Bourne shells. These are the names we use in the discussion in this chapter. (Sometimes we call them **shell startup files**.) Table 3-1 summarizes which scripts are run at login and which are run when a new shell starts up.

You can issue the C shell command **printenv** at any time to check your environment variables. For Bourne and Korn shells, issue the command **set**. For Aegis shells, issue the DM command **env** *variable_name*.

Table 3-1. Shells and Files that Run with Them

Shell	File Run if Started by Login	Always Runs File	Example Filename
Bourne (/bin/sh)	~/.profile	$SHENV	~/.shrc
C (/bin/csh)	~/.login	~/.cshrc	
Aegis (/com/sh)	~/user_data/sh/login	~/user_data/sh/startup	
Korn (/bin/ksh)	~/.profile	$ENV	~/.kshrc

3.1.2 Quick Hints

If you're familiar with setting environment variables, find your shell type in this section; you don't need to read the rest of Section 3.1.

Aegis Shells

```
DISPLAY := "hostname:0"
export DISPLAY

csr . . . /usr/bin/X11
```

Bourne and Korn Shells

```
DISPLAY=hostname:0
export DISPLAY

PATH=/usr/bin/X11 [:pathname2 . . .: pathnamen]
export PATH
```

C Shells

```
setenv DISPLAY hostname:0
set path=( . . . /usr/bin/X11)
```

3.1.3 Setting the DISPLAY Environment Variable

Every time an X client program runs, it must be told which server to connect to. Most clients accept the command–line option **-display**; if this option isn't found, they check the DISPLAY environment variable.

If you don't do anything, your DISPLAY variable may automatically be **:0**, which means that your clients will run and display okay on your workstation but can not run or display remotely. In other words, the client and server will communicate using UNIX Domain Sockets. This default environment variable setting is created in various files in **'node_data**, depending on the kind of workstation you have: **startup**, **startup.1280bw**, **startup.1280color**, **startup.191**, or **startup.color**.

Because of X's client/server configuration, the server and client can be on separate computers. You could, for example, be using a DN3500 that is running the X server. If you then logged in remotely to another system (assuming the proper TCP connections) and started an X client there using the **-display** option, that client could do a computationally intensive simulation on that other system but display its graphics results in a new window on your DN3500. (Most of the time, however, the client and server are on the same workstation.)

The form of the DISPLAY environment variable is

hostname:0

where *hostname* is the TCP/IP hostname of your workstation. Usually, the hostname is just the name of your workstation without the // prefix. However, we suggest that you use the **hostname** command to verify that the current hostname is your workstation's name.

The full syntax for the DISPLAY environment is actually

hostname:*display*[.*screen*]

where *display* is the display number (typically 0), and *screen* is the screen number (the default screen is used if this is not specified).

You can set the DISPLAY variable in several ways, depending on which shell you're running, and whether you want the variable to be set for anyone who logs in at your workstation.

C Shells

Set DISPLAY for a C shell by putting the line

```
setenv DISPLAY hostname:0
```

in your **.login** (or **.cshrc**) file. The variable will not take effect until you log out and log in again, **source** the file, or start a new C shell (if you put the line in **.cshrc**).

Bourne Shells

Set DISPLAY for a Bourne shell by putting the following lines

```
DISPLAY=hostname:0
export DISPLAY
```

in your **.profile** or shell startup (e.g., **.shrc**) file.

Korn Shells

You can set DISPLAY for a Korn shell the same way you do for a Bourne shell, except that typically you might want to use **.kshrc** rather than **.shrc** for your shell startup filename. Unlike the Bourne shell, you can set and export DISPLAY with one line:

```
    export DISPLAY=hostname:0
```

Aegis Shells

If you want to set DISPLAY on a per-login basis, add the following lines to a **user_data/startup_dm.***xxx* or **/sys/dm/startup_login.***xxx* file:

```
    DISPLAY := "hostname:0"
    export DISPLAY
```

All Logins

You can also set DISPLAY globally for the entire workstation so that anybody who logs in will inherit the variable, regardless of that user's system type. Add the line

```
    env DISPLAY 'hostname:0'
```

to your workstation's **'node_data/startup.***xxx* boot startup file. (It has to be in this startup file because you can't use the **env** command once the first user process has executed.) Note that this DISPLAY variable may be overridden by a different setting in a **.login** or **.profile** file.

Setting DISPLAY for Individual Clients

Before running any client, you can set the DISPLAY variable by using the appropriate Bourne, Korn, or C shell environment command.

You can also specify a display option on your command line when you run most clients, by adding

```
    -display hostname:0
```

3.1.4 Setting the Correct PATH

You must add the directory

/usr/bin/X11

to your default search PATH. The **bin** directory contains all X binaries and X clients, except the **Xapollo** server, which, although it's in **/etc**, has a link from **/usr/bin/X11.**

In the C shell, your default PATH is usually set in your **.login** file, but you can set it in your **.cshrc** file. The format is

```
set path = (pathname [pathname2 . . . pathnamen])
```

where the *pathnames* are separated by spaces.

In the Bourne or Korn shell, the default path is set in your **.profile** file. (If you prefer, you can also set it in your shell startup files, e.g., **.shrc** or **.kshrc**.) The format is

```
PATH=pathname[:pathname2: . . .: pathnamen]
export PATH
```

where the *pathnames* are separated by colons and provide for your normal search paths (for example, **~/bin**, **/usr/bin**, and so on).

3.2 Running X Clients

This section provides general information for running X clients. Note that any Open Dialogue program is an X client that can be run under the Domain/X11 server. See Chapter 4 for information on using resource files to specify fonts, colors, and so on.

3.2.1 General Rules for Running X Clients

You should usually run X clients in the background with the & character, so that the window from which you run them will still be usable for other purposes, for example

```
xterm &
```

(Some clients, such as **xset**, **xsetroot**, and **xmodmap**, do their work quickly and return. They don't need to be run in the background.)

Listing Your Processes

Use the **ps** command with an **–ax** option to list all the processes running on your workstation:

```
ps -ax
```

Notice that each process has a process ID number. (If you're using a SysV environment, use **–e** for the option, as in **ps –e**.)

Stopping Processes

Use the **ps –ax** command to list your processes, and then stop a particular one by using its *process_id* in a **kill** command, as in

kill *process_id*

The display associated with that process will disappear, and you'll see a message confirming it. For instance, if you **kill** the process ID associated with the **xclock** client, the clock will disappear, and, after the next prompt, you'll see the message "terminated xclock."

(You can't kill an **xterm** in this way because **xterm** does a **setuid root**—you'll have to use an exit key such as ^D. Likewise, you can't kill the server if it is started in **/etc/rc**, as described in Sections 1.1 and 2.2.3, unless you're logged in as root.)

3.2.2 Displaying Your Clients

Each client has command line options appropriate to itself. (See the individual man pages by using the **man** command.) Most clients, however, share a small set of options (see **man X**). The simplest are

–rv	Create the window in reverse video.
–bw *n–pixels*	Use a window border *n–pixels* wide.
–fn *fontname*	Use the screen font specified (see Section 4.3.3)

The most important is

–geometry *WIDTHxHEIGHT*{+|–}*xoff*{+|–}*yoff* which specifies the position and size of the window. *WIDTH* and *HEIGHT* are the size of the window in pixels. (The exception is **xterm**, where they specify the number of columns and lines.) *Xoff* specifies the location of the left or right edge; *yoff* specifies the location of the top or bottom edge.

The x/y offset must be given in pairs. Positive values specify the left and top edges of the client windows; negative values the right and bottom edges. Placement is specified by various combinations:

+xoff+yoff	right and down from the upper left corner
+xoff–yoff	right and up from the lower left corner
–xoff+yoff	left and down from the upper right corner
–xoff–yoff	left and up from the lower right corner

The following example command line

```
xclock -geometry 200x200+50+100
```

starts an **xclock** window that is 200x200 pixels (approximately 2 inches square). From the upper left corner, it is put about half an inch (50 pixels) to the right, and about an inch (100 pixels) down. Use **xlswins** to report on the position of an X window (see **man xlswins**).

When DM owns the root and no **geometry** specification is given, the client assumes a default size and uses +0+0 (the upper left corner).

When X owns the root and no **geometry** specification is given, the **uwm** window manager asks you to indicate where you want the client to go. Do the following to position the window:

1. Start the client in the background. It will appear some seconds later as a flickering outline attached to the cursor, which has changed into a right-angle figure.

2. Move the mouse to position the upper left corner of the client's outline.

3. Indicate the size of the client, usually by pressing a mouse button:

left	Default-sized window
center	Variable-sized window (the size changes as you move the mouse and freezes

	as you release the button)
right	Large window extending down from the cursor to the edge of the screen

3.2.3 Other Options on the Command Line

You can specify other resource options on the command line when you run a client. This will let you tailor a specific client while keeping most of your clients to the specifications in your **.Xdefaults** file (which is explained in Section 4.4). The **geometry** specification in the preceding section is one example of a resource specification. Other resources you might want to specify can use an **xrm** or a **name** specification.

The -xrm Option

The various resources that are honored by a client can be specified with an **-xrm** option on your command line, as in

client **-xrm** *'resource_spec_string'*

where *resource_spec_string* (which belongs inside single quotes) is a valid specification as explained in Section 4.4, for example

```
xterm -xrm 'xterm*Foreground: blue' &
```

The -name Option

You can also name a particular client (if it supports the **-name** option) so that it uses resources that begin with that name. For instance,

```
xterm -name big_font &
```

will create an **xterm** with the resources specified for **big_font**.

3.2.4 Running Remote Clients

As noted earlier, clients do not have to be running on the same system as the X display server. For example, one client can be running on a high-speed mainframe, collecting telemetry data from a space shuttle and displaying graphs on 20 different displays running X servers. The workstation to which the display is attached is called the **server workstation**; the workstation on which the client is running is called the **remote workstation**.

Before you run remote clients,

- Both the server workstation and the remote workstation have to be running a TCP/IP server.

- The hostname and its TCP address must be in **/etc/hosts**.

- The server workstation has to have given the remote workstation permission to create windows on its display. This security mechanism keeps unsolicited windows from popping up on your display.

  If you need to give permission temporarily for just one session, use the **xhost** command:

  ```
  xhost +otherhostname
  ```

  To give permission permanently, edit the **/etc/X0.hosts** access control list file (which is actually a link back to the file **'node_data/etc/X0.hosts** on your local workstation). This file is on each workstation so that security can be controlled on a per-workstation basis. There must be only one hostname per line, with no leading or trailing blanks. Note that the more hosts you add to this list, the longer it will take the **Xapollo** server to initialize, as noted in Section 2.3.2.

- The client on the remote workstation must correctly name the display on which it wants to create the window. This means either that the DISPLAY environment variable on the remote workstation has to be set to point back to the server workstation, or that a **-display** *hostname*:**0** option has to be given on the client's command line.

- The Apollo workstation where the client will run must have an **/etc/sys.conf** file that lists the appropriate run-time library or libraries (see Section 5.2.2.)

The following exchange shows a user on a workstation named **a** typing into an **xterm**. The **rlogin** command logs the user into a VAX on the local net, where he starts up an **xload** client that monitors system activity on the VAX and displays it on the **a** workstation.

```
a% xhost +vaxnumber2          (give vaxnumber2 permission)
a% rlogin vaxnumber2          (remote login to the VAX)
login: user                   (VAX's login prompt)
Password:                     (VAX's password prompt)

    Welcome to the Local VAX Ultrix Server!

vax% setenv DISPLAY a:0       (set DISPLAY to node a)
vax% xload &                  (monitor VAX load and show
                               it on the a workstation)
```

3.3 Important X Clients

This section provides an explanation or synopsis of the more important X clients (excluding games and simple demos such as **ico**, **xcalc**). In some cases, we'll describe several of the clients in the same section, so they won't be exactly in alphabetical order. At all times, the man pages form the definitive word for the clients.

See Table 3-2 for a list of the clients provided by this version of Domain/X11, and note the column for sections of the manual where these clients are discussed. Some are not explained in this manual but only in the man pages; others are just briefly noted in Chapter 1.

3.3.1 Client Images, xdpr, xpr, xwd, and xwud

Xwd lets you put a window image into a file. You can use the resulting cursor or the options **-name**, **-id**, or **-root** to specify the window, and you can redirect the output to a file using >*filename* or just **-out** *filename*. Once the image has been stored, you'll hear two beeps.

When you've got the image, you can redisplay it with **xwud**, the screen "undumper." Use the **-in** *filename* option to specify the file.

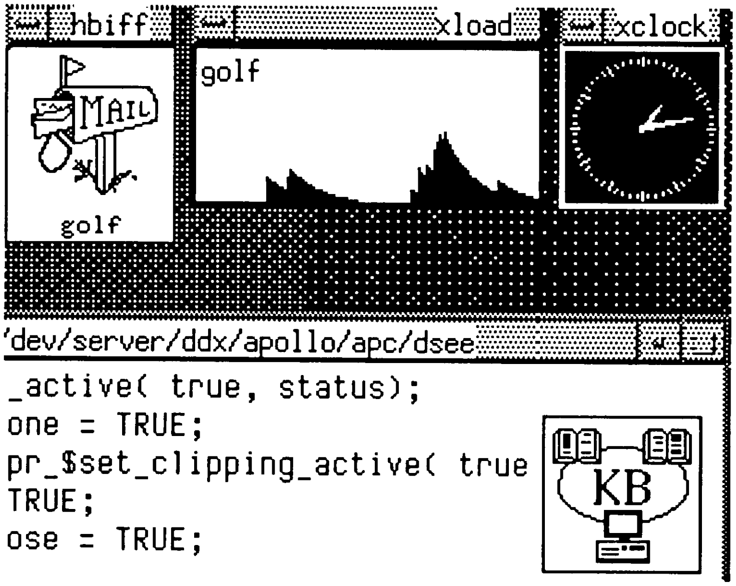
If you use the **-display** *hostname*:**0** option when you pick up the window with **xwd**, you can choose one from someone else's screen. This can be a lot of fun. (If you specify **-root**, you'll get the entire window.)

Table 3-2. X Clients that Aren't Games

Client	Purpose	Section
atobm	converts from ASCII to bitmap	3.3.2
bdftosnf	converts **bdf** fonts to **snf** fonts	3.3.2
bitmap	edits a bitmap image	3.3.2
bmtoa	converts from bitmap to ASCII	3.3.2
mkfontdir	builds **font.dir**	4.3.3
resize	resizes clients that ignore SIGWINCH	3.3.8
rgb	builds color database from **rgb.txt**	man
showsnf	displays **snf** font info	3.3.2
startx	runs **xinit** from a shell script wrapper	2.3.4
uwm	"Universal Window Manager"	2.5.2
xbiff	flags mailbox receipts	1.6.2
xclipboard	accumulates text in a clipboard	3.3.8
xcutsel	interchanges between cut & selection	3.3.8
xdpr	dumps an X window to the printer	3.3.1
xdpyinfo	displays information about the server	man
xedit	edits simple text	1.6.2
xev	displays events for a window	4.1.3
xfd	displays fonts	1.6.2
xhost	allows server access to a workstation	3.2.4
xinit	initializes the X window system	2.3.4
xkill	kills a client by its resources/window	man
xload	displays CPU load average	man
xlsfonts	lists fonts known to the server	1.6.2
xlswins	lists the X window tree	man
xmag	magnifies parts of the screen	3.3.3
xman	displays manual pages	3.3.4
xmh	interfaces to the MH mail system	1.6.2
xmodmap	modifies key mappings	3.3.5
xownroot	toggles X/DM ownership of root	3.3.6
xpr	prints an X window dump	3.3.1
xprop	displays properties for an X window	4.4.5
xrdb	updates the server's resource database	3.3.7
xrefresh	refreshes all or part of screen	man
xset	sets user preferences (display, etc.)	4.3.3
xsetroot	changes root window appearance	man
xterm	emulates a VT102 terminal	3.3.8
xwd	dumps image of an X window to a file	3.3.1
xwininfo	displays window information	1.6.2
xwud	displays window images	3.3.1

Alternatively, you can use the **-display** *hostname*:**0** option with **xwud** and the image will appear on someone else's screen—assuming you are authorized in **/etc/X0.host** to display windows there. (It goes away as soon as a key or button is pressed on that workstation, but if the other user is waiting for a process to complete, it might be a while.) Figure 3-1 shows a corner of a screen we captured from someone else using the command

```
xwd -out golf.screen -root -display golf:0
```



```
_active( true, status);
one = TRUE;
pr_$set_clipping_active( true
TRUE;
ose = TRUE;
```

Figure 3-1. **xwd -out golf.screen -root -display golf:0**

Printing the Image

If you want to print the image, you'll need a PostScript printer. Run the **xpr** client to convert your file to PostScript:

```
xpr -device ps -output filename.ps  filename
```

Notice that we've used **.ps** for an extension: you can use that or anything else you might want. Now that you've got the PostScript file, you can print it. We've found that the following Aegis command works nicely:

```
prf -pr postscript_printer -trans filename.ps
```

If you don't have Aegis print facilities, then your system administrator will have set up a Berkeley spool site according to the directions in Chapter 6 of *Managing BSD System Software*, or a SysV site according to *Managing SysV System Software*. For a BSD system, the **/etc/printcap** file will specify input and output filters that will pass your PostScript file without changing it. For a SysV system, models will have been set up to do the same thing. Then, you can use the following commands:

```
lpr -Pprintername filename.ps      (BSD)
lp -Dprintername filename.ps       (SysV)
```

Summary

To capture and print a screen image, enter the following commands:

1. xwd -out *filename*
2. xwud -in *filename*
3. xpr -device ps -output *filename*.ps *filename*
4. prf -pr *printername* -trans *filename*.ps

   or

   lpr -P*printername filename*.ps (BSD)

   or

   lp -D*printername filename*.ps (SysV)

More

The **xpr** "printing" client has a lot of options that let you specify scale, density, height, width, orientation, headers, etc. See the man page and use what is appropriate for your needs. Note that we've shown **xwud** in Figure 5–5 just so you can check your image, and that we recommend you ignore **xdpr** for now.

3.3.2 Bitmaps

This section very briefly explains how to use the clients **bdftosnf, showsnf, atobm, bmtoa,** and **bitmap** to create and modify icons and bitmaps. Again, this will be just an introduction. See the man pages for each of these clients for more information.

Getting an ASCII Version of a Bitmap

Let's say you've created an ASCII version of a bitmap, as in Figure 3–2, where all the blank pixels use a dash (–) and all the lit pixels use a pound sign (#).

```
----------------###---------------------
-----------#####--#---------------------
---------##-------#---------------------
-------##---------#---------------------
-----##---------##----------------------
--###--------###------------------------
--#--------#####################--------
--#-----------------------------#-------
--#-----------------------------#-------
--#-----------------------------#-------
--#---------------##############--------
--#--------------#----------------------
--#---------------######----------------
--#------------###------#---------------
--#------------#-#------#---------------
--#------------###------#---------------
--#---------------######----------------
--#------------###------#---------------
--###----------#-#------#---------------
-----##--------###------#---------------
-------##--------#######----------------
---------#########----------------------
```

Figure 3–2. ASCII Version of a Bitmap

You can create these ASCII versions of bitmaps either with a text editor, or you can use the **showsnf** client if you've got the icon in **server normal format** (see Section 4.3.3 for more on fonts).

For our example, we ran **edfont** on some icons stored in someone's **user_data** directory, then we saved the font in **bdf** format (which is also described in Section 4.3.3). From there, we converted the icons to **snf** format with the command

```
bdftosnf our_icons > our_icons.snf
```

Then we ran **showsnf** on them

```
showsnf -g our_icons.snf > /tmp/icons.array
```

and opened the output file with a text editor until we found the ASCII version of the icon we wanted to use for an example here. We cut out the appropriate lines and pasted them into **hand.ascii**.

Converting between ASCII and Bitmaps

The two clients **atobm** and **bmtoa** convert from ASCII bitmaps to "real" bitmaps and back again. Once we had **hand.ascii**, we used the command

```
atobm hand.ascii > hand.bitmap
```

(Of course, you can use your own names and extensions.) The resulting bitmap can be edited with the **bitmap** client, and then, if you wanted, for example, to send it to someone on the net, you could convert it back to ASCII using

```
bmtoa bitmap_file > ascii_file
```

Starting the bitmap Client

Run **bitmap** with the command

```
bitmap bitmap_file &
```

We don't have room on these pages to display all of the result, so we'll only show enough of it so you'll recognize it. When your **bitmap** window comes up, you'll see a large grid, as in Figure 3-3, containing the enlarged pixel image of your bitmap or icon. In the lower right corner, you'll find the bitmap displayed its actual size, along with an inverted version (we've put this in the canvas area). Every time you edit or change a pixel on the canvas, these actual-size versions will change, too.

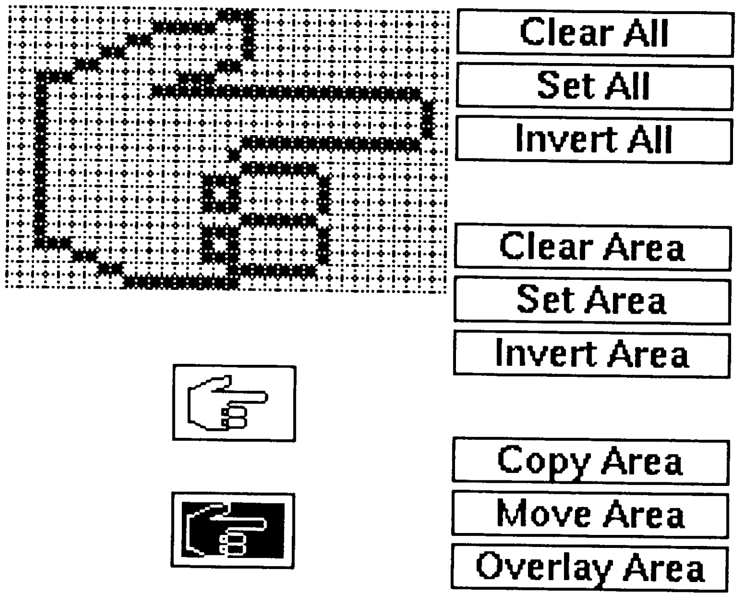


Figure 3-3. The **bitmap** *Canvas*

Using bitmap Commands

The mouse buttons work as described in Table 3-3. The command boxes are generally self-explanatory, *but note that there is no* **undo** *function*.

To define an area, you'll see an "upper-left-corner" cursor. Press and hold any button to accept that position, and then you'll see a "lower-right-hand" cursor that you can sweep down to the lower right hand corner of the area that will be defined when you release the button. If you've chosen a copy, move, or overlay command box, you'll then see *another* upper-left-corner cursor that you can move to where you want the area to be copied, moved, or overlaid.

Table 3-3. Mouse Buttons for **bitmap**

Button	**Purpose**
Left	Draw a pixel: change one or more cells from background to foreground color
Middle	Invert a pixel color: change one or more background colored cells to foreground, or from foreground to background.
Right	Clear a pixel: change one or more cells to the background color.
(On a monochrome display, the background color is white and the foreground color is black.)	

To draw lines and circles, select the appropriate command box and move the resulting round cursor (●) to where you want the start of the line or the center of the circle, then click any button, move the next round cursor to the end of the line or radius, and click any button.

Bitmaps as Cursors

If you want to use your bitmap as a cursor, *make sure it's only 16 x 16 squares*. (Our current example won't work—it's on a 22 x 44 grid.) Then, you need to make a mask of your bitmap. The cleared areas on the bitmap will be opaque and show up against the background, provided that the same areas are inverted on your bitmap mask. (See Figure 3-4.)

If you store your bitmap and mask in your working directory, you can use a command similar to this to set your root window cursor:

```
xsetroot -cursor ~/circle.bitmap ~/circle_mask.bitmap
```

If you look at the bitmaps themselves, you'll see that they're C source code. The man page for **bitmap** tells you how to include this code in your program when you make a cursor for your own X client (see the section titled "Using Bitmaps in Programs" in the man page).

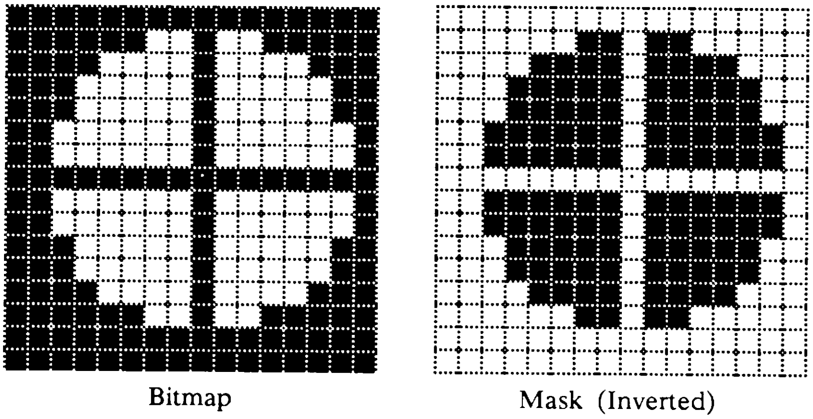


Figure 3-4. Bitmap and Mask

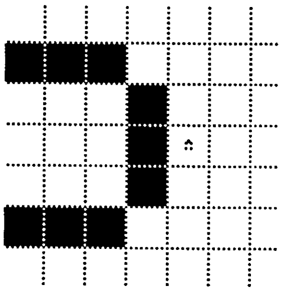


Figure 3-5. The Hot Spot

Hot spots are used by cursors to show the location of the pointer. The hot spot on a cross–hair cursor, for instance, is in the center; for an arrow, it's at the tip, and so on. If you want your bitmap to be useful as a cursor, you'll want to set the hot spot. For our example, we've chosen to set it in the first pixel next to the index finger of our hand. You'll notice, even in Figure 3–5 which is significantly enlarged, that the hot spot shows up as a very small diamond. If you're using someone else's bitmap and can't find the hot spot, just clear it and then put it where you want it.

3.3.3 Magnifying Images with xmag

The **xmag** client lets you magnify parts of the screen. By default, you'll see a 64–pixel square image that is five times larger than the original, if you issue just the command

```
xmag
```

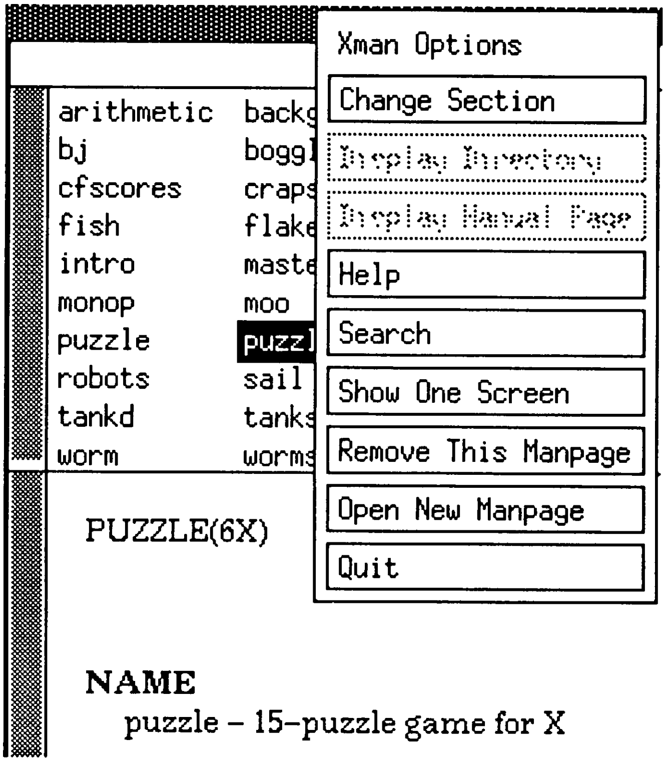
and then position the cross–hair cursor to the center of the area that you want magnified and click the first mouse button.

You can then move the arrow cursor into the **xmag** window. Press and hold down the first mouse button to see the x and y coordinates, the bitmap bit setting, and RGB color value of the pixel you've chosen. (See Figure 3–6, where we've magnified the Interleaf heading for Section 3.3.3.) To quit the **xmag** client, press one of the keys q, Q, or ^C in the magnified window.

Figure 3–6. An **xmag** Image

3.3.4 Reading Man Pages with xman

We've included **xman** in our distribution of Domain/X11, but it only works with BSD man pages, not SysV man pages. Basically, **xman** gives you a point-and-click method for reading man pages. You can display the directory for different man page sections and click on a particular man page name to see it. These include any man pages found in the directory specified by your MANPATH environment variable (it defaults to **/usr/man/man1**), plus you can change from section to section.



```
arithmetic   backg
bj           boggl
cfscores     craps
fish         flake
intro        maste
monop        moo
puzzle       puzzl
robots       sail
tankd        tanks
worm         worms
```

PUZZLE(6X)

NAME
puzzle – 15-puzzle game for X

Figure 3-7. A Page from **xman**

The first **xman** window contains three buttons: Help, Quit, and Manual Page. Manual Page displays a formatted version of the **xman** man page. Move the pointer into the title bar of the **xman** man page window and press the left mouse button to see the **Xman Options** pulldown menu. Release the button and notice that the borders of the options highlight as you move past them. Click the left button when you've highlighted the option you want. (See Figure 3–7, which shows what happens when you select the Show Both Screens option: whatever man page you click on in the top "screen" will be displayed in the bottom "screen.")

3.3.5 Modifier Keys and xmodmap

You should run the **xmodmap** program, after the server has started, in order to define "modifier" keys (**Control**, **Shift**, and **Meta**) that will work with the mouse. Many X clients (such as the **uwm** window manager, **xterm** menus, and **gnuemacs**) need modifier keys.

Modifier keys and **xmodmap** are discussed in greater detail in Section 4.2. Since **xmodmap** runs quickly and then exits, you won't need to run it in the background.

Keyboard 2 users need to define all three keys (**Control**, **Shift**, and **Meta**) because their white CTRL and SHIFT keys modify only white keys, not the mouse. Typing

```
xmodmap  /usr/lib/X11/keyboard/xmodmap.kb2sample
```

defines <F1> as **Control**, <F2> as **Shift**, and <F3> as **Meta**. White keyboard 2 keys cannot be used as modifiers for the mouse (<CTRL> and <SHIFT> continue to work for white keys).

Keyboard 3 users start out with the white CTRL and SHIFT keys working properly with the mouse, and they only need to run **xmodmap** if they want a **Meta** key. To define <F0> as **Meta**, run

```
xmodmap /usr/lib/X11/keyboard/xmodmap.kb3sample
```

NOTE: Keys redefined with **xmodmap** lose their DM key definitions, whether they are typed in X or DM territory. You must decide which keys you can do without. The DM **kd** (key definition) command does not show changes made through **xmodmap**, and once **xmodmap** makes a change, the DM knows nothing about that key until the workstation is rebooted.

3.3.6 Switching Root Ownership with xownroot

You can switch root ownership between the DM and your X window manager without stopping the **Xapollo** server. Use the **xownroot** command (type **man xownroot** to see the man page). As with **xmodmap**, don't run **xownroot** in the background. Be aware that *only one window manager should be running at a time*, and window managers don't start up by themselves when you switch root ownership.

Changing to DM Root Ownership

To change to DM root ownership, you'll have to turn off your X window manager (e.g., **kill** the process), issue the **xownroot -off** command, and then turn on DM window management (**wmgr -on**).

Changing to X Root Ownership

To change to X root ownership, turn off the DM window manager (**wmgr -off**), issue the **xownroot -on** command, and then start an X window manager (e.g., issue the command **uwm &**).

3.3.7 The xrdb Resource Database Client

The **xrdb** client, normally run from your X startup file, manages the resource database used by the X server. It has the general syntax of

xrdb [*option* . . .] [*filename*]

where *filename* specifies the name of a file that contains resource specifications (see Section 4.4), and *option* is typically **-load**, **-query**, **-merge**, or **-edit**. A typical **xrdb** command is

```
xrdb -load .Xresources
```

The resource manager (used by the Xlib routine **XGetDefault(3X)** and the X Toolkit) uses the RESOURCE_MANAGER property (actually, a data structure) to get user preferences about color, fonts, and so on for applications. Having this information in the server allows for dynamic changing of defaults without editing files.

For compatibility, when **xrdb** has not been run or the property has been removed (by the **-remove** option), the resource manager will look for a file called **.Xdefaults** in your home directory.

The -merge Option

As we saw in the **xinitrc** script, the most common option is **-merge**. This option indicates that the input should be merged with, instead of replacing, the current contents of the RESOURCE_MANAGER property. Since **xrdb** can read the standard input, you can either specify an input filename or just type multiple resource properties, for example

```
xrdb -merge
xterm*scrollBar: true
xterm*Foreground: blue
^D
```

The -query Option

If you've used **xrdb** to load options, you can see what they are:

```
xrdb -query
```

For our **xterm** example, if that's all that has been set, you'd see

```
xterm*Foreground:   blue
xterm*scrollBar:    true
```

The -edit Option

You can use the **-edit** option to save the current value of the RESOURCE_MANAGER property in a file. This lets you save, for instance, someone else's resource settings if you happen to like them. The general format for this option is

xrdb -edit *output_file* [**-backup** *extension*]

where the **-backup** option lets you name an extension to be added to the name of the *output_file*, if one exists (this keeps your **-edit** option from overwriting an output file). For example,

```
xrdb -edit ~/.Xdefaults -backup bak
```

will create the file **~/.Xdefaults.bak** from your current **.Xdefaults** file and will save the new property information in **.Xdefaults**.

3.3.8 The xterm Client

A terminal emulator, **xterm** creates an 80-character by 24-line "screen" by default and uses the SHELL environment variable to start a shell running in it. An **xterm** can emulate a VT102 text-oriented terminal or a Tektronix 4014 graphics terminal, each of which has its own menu.

Table 3-4. The **xterm** *Menu via* **^Button1**

Menu Entry	**Purpose**
Secure Keyboard	Ensure that all keyboard input is directed only to the **xterm**.
Visual Bell	Flash the **xterm** when a ^G is received, rather than ring the bell.
Logging	Send all terminal output to a log file (**XtermLog**.*xxx* in your current directory) as well as to the screen.
Redraw	Redraw the **xterm**.
Suspend program	Send a SIGSTP signal.
Continue program	Send a SIGCONT signal.
Interrupt program	Send a SIGINT signal.
Hangup program	Terminate the **xterm** with a SIGHUP.
Terminate program	Send a SIGTERM signal.
Kill program	Terminate the **xterm** with a SIGKILL.
Quit	Terminate the **xterm**.

The xterm Menu

You can bring up the basic **xterm** menu by pressing the CTRL key and the left mouse button at the same time, then releasing the CTRL key when the menu appears. If you then sweep down the entries, they're highlighted as you pass them. The first few entries toggle on and off when you release the mouse button on any of them, and they display a check mark when they're on. (See Table 3-4.)

The Modes Menu

The Modes menu (see Figure 3-8) comes up when you press the CTRL key and the middle mouse button. (The middle mouse button works like the left button does on the basic **xterm** menu.)

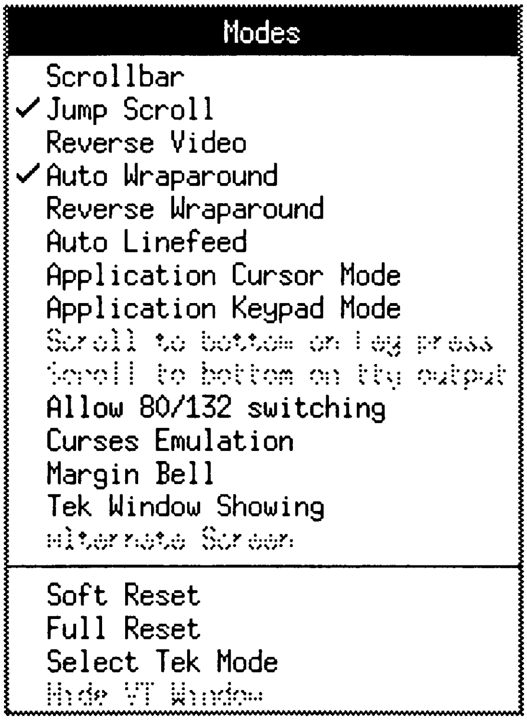


Figure 3–8. **xterm**'s *Modes Menu via* **^Button2**

The stippled options indicate features that aren't available given your current settings: if the scrollbar is turned off, you won't see the scrolling options on the Modes menu, for example.

To pop up the Tektronix window, select "Tek Window Showing." Then you can use that window by choosing the option "Select Tek Window." You can also hide your VT102 window at that time.

The Tektronix Menu

The Tektronix menu (see Figure 3–9) comes up when you press the CTRL key and the middle mouse button when your cursor is inside the Tektronix window. (The middle mouse button works like the left button does on the basic **xterm** menu.)

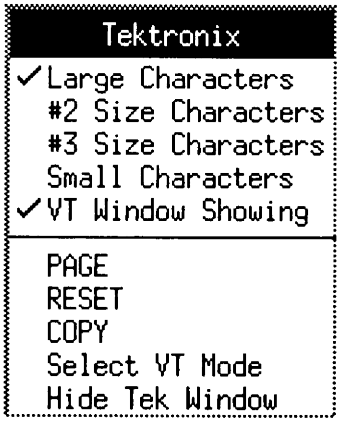


Figure 3–9. The Tektronix Menu

Selecting and Pasting Text

To select text in an **xterm**, **xedit** window, or other text-oriented X window, follow these steps:

1. Position the cursor at the start of the text.

2. Select the end of the text by doing one of the following:

   - Clicking and holding the left mouse button and then moving the pointer to the end of the region you want.

   - Clicking twice and holding the left mouse button to highlight the word the cursor is on, and then moving the pointer to the end of the region you want.

   - Clicking three times and holding the left mouse button to highlight the current line and then moving the pointer to the last line of the region you want.

   - Moving the cursor to the end of the text you want and pressing the right mouse button.

3. Accept the highlighting by releasing the mouse button.

You've now highlighted a range of text that remains highlighted to show you it is in the paste and cut buffers (named PRIMARY and CUT_BUFFER0, respectively). **Xterm** selections automatically go to both buffers; other clients may just put this into the PRIMARY buffer. We'll discuss this a little more in the next two subsections.

To paste the text, select one of these options:

- Move the cursor as appropriate to your editor's input expectations.

- Close the current file to return to the command line prompt in the same window.

- Move to the command line prompt in another window.

And then press the middle mouse button.

Notice that the original highlighting remains to show you what's in the buffer—if you haven't closed the text file. Depending on the client, you have either pasted text from the PRIMARY selection buffer, or from CUT_BUFFER0 if the primary buffer is empty. Note that you can also paste over existing text in any editor with an overwrite mode.

To deselect the text, move the pointer off the highlighted area and press the left button. The highlighting goes away, but the text remains in the buffer(s).

Using xcutsel to Copy Buffers

In Release 2 clients, there was only a cut buffer. At Release 3, many clients, such as **xterm**, also implement a primary buffer. If you want to paste between Release 2 and Release 3 clients, you'll need to use the **xcutsel** client to switch to the right buffer. As you can see in Figure 3-10, **xcutsel** lets you copy from the select (PRIMARY) buffer to the cut (CUT_BUFFER0) buffer, or vice versa. Once you've switched to the right buffer, you can paste from it. (In Figure 3-10, the options on the left are displayed if you haven't loaded any resources at all.)

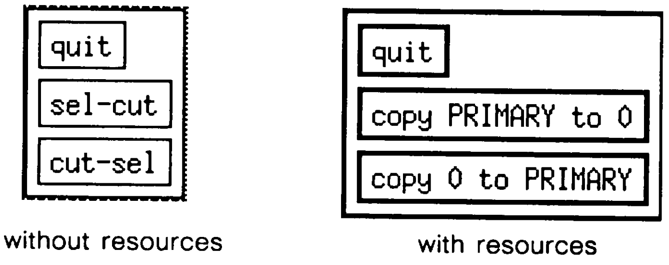


Figure 3-10. The xcutsel *Window*

Using xclipboard to Save Buffers

The **xclipboard** client lets you paste into it and copy from it. The erase button doesn't work, and **xclipboard** can't write its buffer into a file, but the quit button works, and you can accumulate text in the text area by pointing to the area and pasting (your new text enters automatically at the end). See Figure 3-11.

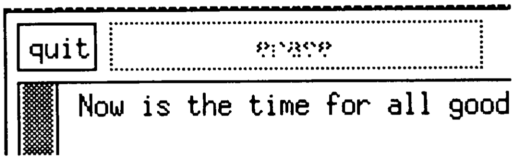


Figure 3-11. The **xcilpboard** *Buffer*

Text can be copied to the clipboard automatically whenever a client asserts ownership of the CLIP_BOARD selection. Examples of event bindings to include in your resource configuration file to use the clipboard this way are

```
*VT100.Translations: #override \
         Button1 <Btn2Down>:      select-end(CLIPBOARD) \n\
         Button1 <Btn2Up>:        ignore()

     *Text.Translations: #override \
         Button1 <Btn2Down>:      extend-end(CLIPBOARD)
```

With these translations, which could be in your **.Xresources** file, you would select text by pressing Button 1, as usual, but when you also pressed Button 2, the text automatically goes to the clipboard.

(See the individual client documentation for how to make a selection and send it to the clipboard. For more information on events and translations, see Vol. 4, *X Toolkit Programming Manual*.)

Resizing an xterm

Before using an **xterm**, some I/O options should be set with the **stty** command. (Type **man 1 stty** for more information.) The minimum options for an **xterm** can be set with the following command line in your **.cshrc** file:

```
stty new crt
```

Your mouse cursor can be anywhere in the **xterm** window and you can still type. (Unlike the DM, X clients often maintain a keyboard cursor that is independent from the mouse cursor.)

You can treat an **xterm** just like a DM input/transcript pad with one exception: you cannot directly run a DM program in an **xterm** that is going to try to "take over" the existing window as is done by programs such as Interleaf with the **-w** option. Start such programs from DM windows or from the "wrapper" routine **dmwin**, as explained in Section 5.5.1. The error message "Attempted operation illegal on this stream (stream manager/IOS)" when you try to run a program from an **xterm** means that you should either run the program with **dmwin** or run the program from a DM pad.

On starting, an **xterm** searches the termcap file **/etc/termcap** (or, in SysV, **terminfo**) for "xterm," "vt102," "vt100," and "ansi," and then sets the TERM and TERMCAP environment variables. (Type **man 5 termcap** for more information on the termcap file.) The default **/etc/termcap** file includes definitions for an **xterm**, so that when an **xterm** is started, it will set TERM and TERMCAP to **xterm**.

You can check the size of an **xterm** by issuing the command

```
stty size
```

On BSD4.3 systems, a program can get size information from the **xterm**. (**More** and **vi** use this method.) They can also read the TERMCAP environment variable to determine the size of the **xterm**. This means that the TERMCAP environment variable must be updated every time the **xterm** is resized. You can make this happen by executing the following commands in the **xterm** when it is resized:

```
set noglob
eval '/usr/bin/X11/resize'
unset noglob
```

We recommend defining an alias in your **.cshrc** that includes the three steps mentioned above. For example, by including the lines

```
alias newsize ´set noglob; \
                eval `/usr/bin/X11/resize`; \
                unset noglob´
```

you can enter the alias **newsize** after resizing an **xterm**, and the environment will be updated to the new size.

xterm Escape Sequences

Appendix E in O'Reilly, Vol. 3, *X Window System User's Guide*, describes the escape sequences that can be used with **xterms**. See also the **man 7 xterm** page. Two common ones are shown below

Esc [? 3 8 h	from VT102 to Tektronix mode
Esc Etx	switch to VT102 mode

You can put these escape sequences into one of your startup or login scripts with **seltek** and **selvt** as aliases:

```
# xterm Tek window
alias seltek ´echo -n line "E[?38h" | tr E \\033´
# xterm VT102 window
alias selvt ´echo -n "EC" | tr EC \\033\\003´
```

Chapter 4

Keyboards, Workstations, Resources

Contents

Chapter 4

Keyboards, Workstations, and Resources

This chapter tells you how to customize your keyboard. It also discusses color, fonts, and additional client resources.

Discussions in previous sections of this manual have assumed that you are using a default keyboard configuration established by the sample starter scripts and command line options. The Domain/X11 **Xapollo** server, however, implements the full individual keyboard customization described in the X11 protocol. You don't have to have <F0> as the **Meta** key on keyboard 3; you can make **Meta** anything you like. You can even redefine the entire keyboard to a nonstandard layout such as Dvorak.

Key names are written in upper case and enclosed in angle brackets when they represent the key as it is engraved on the Apollo keyboard (for example, <RETURN>). In cases where the names for the engraved keys are in upper case but not enclosed in angle brackets, the names are always accompanied by the word "key," as in "the RETURN key." Again, these are the *physical* names as engraved. If there is no name engraved on the physical key, the key is referred to generically in lower case, as in "the space key," or "the left arrow key." When a key is discussed in its X sense or as a keysym, we capitalize just the first letter and bold the name, as in "A modifier key such as **Shift** or **Control** modifies the meaning of a key event."

4.1 Xapollo -K *keyboard.config* Option

You can specify which keys go to X and which to the DM when you start the X server. Fairly common choices are implemented by the sample **-K** file **/usr/lib/X11/keyboard/keyboard.config**. If you don't specify a -K *keyboard.config* when you run **Xapollo**, then all keys are recognized by the X server.

The -K option specifies a keyboard configuration file. This file contains simple one-statement-per-line commands that assign the keys on the keyboard (either keyboard 2 or 3) to either the DM or X. *Xkeysym* in the commands below refers to keysyms that are listed in **/usr/lib/X11/keyboard/kb2.xprkbd** or **/usr/lib/X11/keyboard/kb3.xprkbd**. The configuration commands are

quit *Xkeysym*

The **quit** subcommand names a key that will exit the X server when pressed in **X territory** (that is, in any X window listening to keypress events), or in the background root window when X owns the root. (Most X clients, for example, **xterm**, take keypress events. A very few—**xclock** is one—do not.)

The main purpose of the **Quit** key is to provide a way to kill a borrow-mode X server and return to the DM. (The key will work in any server mode, however.) There is no default **Quit** key.

exclude *Xkeysym*

The **exclude** command indicates that the specified *Xkeysym* is excluded from X and assigned to the DM.

When the X server initializes, it takes all keys on the keyboard by default. If you press a key in X territory, then X gets the first chance to process that keyboard event. Only in the rare case of clients—like **xclock**—that do not need to process keypress events does the event pass through to the parent window—which might be the background owned by X or the DM.

This behavior is in keeping with the idea that X should be able to run on Apollo systems as though it were the only window system present, and that clients should be able to assume they can use the whole keyboard. Practically, though, there are keys you want the DM to receive, even if they are pressed in X territory. For example, if the DM owns the root and is handling window management, you probably want keys such as POP, MOVE/GROW, CMD, and NEXT/WNDW to work in X windows.

!

The exclamation point is a comment character (as used in **xmodmap**). It must appear in column 1 in order to denote that the line is a comment.

The **/usr/lib/X11/keyboard/keyboard.config** file is a sample keyboard configuration file that includes all non-white-key X keysyms on Apollo keyboards 2 and 3. You should copy this to your home directory and customize it as you see fit. A reasonable choice of DM keys is already listed as "excluded," so those keys go to the DM no matter in which territory they are pressed. (A **Quit** key command is commented out: note that you have to press **Ctrl-Shift** at the same time as your **Quit** key, e.g., **Ctrl-Shift-F9**.)

Note that these are X keysyms, not DM key names. Also note that case is significant in X keysym names. For example, "A" and "a" are two different, valid keysym names. The **Pop** keysym is "Pop," not "POP" or "pop."

Some keys have two keysyms. For example, <MOVE/GROW> is represented by the keysyms **Grow** *and* **Move**. Specifying either keysym works. It is not possible to "split" a key between the DM and X so that one window system gets the unmodified key and the other gets it modified (for example, with the **Shift** key held down).

To see a complete list of keysyms, including white keys, look at the files **/usr/lib/X11/keyboard/kb2.xprkbd** and **/usr/lib/X11/keyboard/kb3.xprkbd**. (You can also use the **xmodmap -pk** command while the X server is running, but any excluded keys won't show.)

4.1.1 Keyboard Partitioning: An Explanation

A **-K** option to the **Xapollo** server is necessary because there are two window systems (X and the DM) and only one keyboard. With the X server running in share mode (s+), the screen becomes divided into **root** (the background "root" window) and **territory** (all the windows displayed on the root). Ownership of these is as follows:

- The root belongs to the DM if you specified the **r-** option (or to X if the **r+** option) when you started the server.
- The territory either belongs to X (in an X window) or to the DM (in a DM window).

Territory determines which window system gets an input event first. (An input event is a key pressed or released on the keyboard, a mouse button pressed or released, or a mouse movement.) For example, if you type an "a" in an X window, like an **xterm**, then **xterm** gets the "a"; while if you type it in a DM window, the DM gets the "a." (Of course, the X client or the DM might not choose to process a particular input event.)

However, there are exceptions to this "territorial imperative." Apollo keyboards have a number of keys engraved with DM window management functions. Keysyms for these functions are not part of the recognized X standard set as defined in Appendix A of the *X Window System Protocol, X Version 11* and in the "keyboard" set in the header file **/usr/include/X11/keysymdef.h**, so Apollo makes them known to the server by means of a vendor-specific keyboard header file called **/usr/include/X11/ap_keysym.h**.

On one hand, you may want these Apollo-specific keys to continue to have their DM meaning. For example, when X owns the root and an X client is used as the window manager, it's desirable and natural for the POP key to raise an X window.

On the other hand, many of the keys on the keyboard correspond to valid keysyms such as **F0** through **F9**, **Help**, and **Redo**. The **Help** key pressed in an X client can mean either of the following:

- Its original DM meaning
- A way for the X client to provide help to its user

But there can't be two interpretations, so the question arises: which window system owns which keys when?

Early Apollo borrow mode servers from the MIT tape (X11 Releases 1 and 2) had keyboard partitioning hardcoded into the server. X took the Pop, white, F0 through F9, the UpBox and DownBox, and the arrow keys. The DM got the rest.

For the shared Domain/X11 server, keyboard partitioning is left up to the user. By default, if you don't specify a **-K** *keyboard.config* file, X gets all the keys. Our sample **keyboard.config** file gives you a reasonable partitioning, or you can copy it into your home directory from **/usr/lib/X11/keyboard** and edit it.

4.1.2 X Keysym Names

The files **/usr/lib/X11/keyboard/kb*n*.xprkbd** show all Apollo keysyms. You can also use the **xmodmap -pk** command, while the server is running, to list all the current keysyms (except those excluded in a -K *keyboard.config* file).

A sample listing from part of **kb2.xprkbd** follows:

```
0x99      0xffc3        F6
0x9a      0xffc4        F7
0x9b      0xffc5        F8
0x9c      0xffc6        F9
0x9d      0x1000ff0f    Pop
0x9e      0xff66        Redo
0x9f      0x1000ff10    Read
0xa0      0x1000ff11    Edit     0x1000ff12   Save
0xa1      0x1000ff13    Exit     0xff69       Cancel
0xa2      0xff13        Pause    0xff6a       Help
0xa3      0xffb0        KP_0
0xa4      0xffb1        KP_1
0xa5      0xffb2        KP_2
```

The first column is the **keycode** presented by the Apollo Input DeMultiplexor (IDM) to the X server. It is the number the key produces when pressed, and it cannot be changed. (Actually, keyboards 2 and 3 have different hardware keycodes, but the IDM creates a uniform set to present to X.)

Up to four pairs of **keysyms** follow the keycode. Each keysym has two parts: the hexadecimal constant and the text representing the character. The file **/usr/include/X11/keysymdef.h** assigns the hexadecimal constants to symbolic names such as **XK_parenleft** for official X keysyms, for example

```
#define XK_Redo      0xFF66    /* redo, again */
#define XK_Menu      0xFF67
#define XK_Find      0xFF68    /* Find, search */
#define XK_Cancel    0xFF69    /* Cancel, stop, abort, exit */
#define XK_Help      0xFF6A    /* Help, ? */

#define XK_Left      0xFF51    /* Move left, left arrow */
#define XK_Up        0xFF52    /* Move up, up arrow */
```

Apollo keysyms have names like **apXK_UpBox**. They are defined in **/usr/include/X11/ap_keysym.h**, for example

```
#define apXK_RightBox          0x1000FF0C
#define apXK_UpBox             0x1000FF0D
#define apXK_DownBox           0x1000FF0E
#define apXK_Pop               0x1000FF0F
#define apXK_Read              0x1000FF10
#define apXK_Edit              0x1000FF11
#define apXK_Save              0x1000FF12
#define apXK_Exit              0x1000FF13
#define apXK_Repeat            0x1000FF14
```

For the -K configuration file and the **xmodmap** command, you use the name without its **XK_** or **apXK_** prefix (for example, use the name **UpBox**, not **apXK_UpBox**).

The pairs of keysyms after the keycodes mean as follows:

- The first pair shows the keysym when the key is pressed by itself. If no keysym pair follows the keycode, then no keysym is defined for that keycode. (Often, there is no corresponding physical key that will produce the keycode.)
- The second pair shows the keysym produced when the key is pressed in conjunction with the **Shift** or **Lock** modifier.
- The third and fourth pair have relevance only for Apollo international keyboards, because they have more than two glyphs per key. This could allow some modifier key to indicate these additional keysyms, but this function is not currently implemented in Xlib.

Since keysyms represent the engravings on the keys, the keysym names exactly match the engraving when the engraving is text (for example, **NextWndw**). Where the engraving is a picture, the text suggests the function (for example, **RightBar**, **LeftBox**). These names must match the ones used in any *keyboard.config* file that you edit.

A few of the keysyms don't match the engraving at all. This is because the key has a well-known function with a well-known X name, and we use the well-known synonym as defined in Appendix A of the *X Window System Protocol* rather than the Apollo variant, as shown in Table 4-1.

Table 4-1. Variant Keysym Names

X Name	Apollo Key
F10	<F0>
Select	<MARK>
Redo	<AGAIN>
Pause	<HOLD>

4.1.3 Finding Keysyms with xev

You can use the **xev** event client to see what keysym is assigned to any particular key on the keyboard. (Because **xev** displays information about all events happening to its window, you'll also see a lot more than you want; just focus on the part we're showing below.) Position the X cursor inside the small window that shows up in the **xev** window and press any key. You'll see the keycode (in decimal) and both hexadecimal and character representations of the keysym. For example, pressing <F0> shows that it is assigned to **Meta_L**:

```
KeyPress event . . .
    state 0x1, keycode 147 (keysym 0xffe7, Meta_L)
```

4.1.4 Repeating Keys

The **r** option in **xset** can be used to control autorepeat. If a preceding dash or the "off" flag is used with **r**, autorepeat will be disabled. If no parameters are given or the "on" flag is used, autorepeat will be enabled.

Note that **xset -r** toggles the set of repeatable keys: it doesn't just set any single key. Also, the **r** option only toggles those keys that have been defined to be repeat keys. Use the command

```
kbm -A alpha,default
```

to make all the alphabetical keys autorepeat. Type **man kbm** and **man xset** for more information.

4.2 Using the xmodmap Client

This section discusses the topic of assigning modifiers and their related keysyms only as it relates to Apollo keyboards. To read more about **xmodmap**, type **man xmodmap**.

4.2.1 Establishing Modifier Keys with xmodmap

The **xmodmap** command is used to establish which keys will be modifier keys. When pressed in conjunction with another key on the keyboard (or the mouse), a **modifier** key modifies the signal that the keyboard or mouse button sends to the X server.

The common modifier keys used in X are **Shift**, **Control**, and **Lock**. Less common are **mod1**, **mod2**, **mod3**, **mod4**, and **mod5**. The **mod1** key, by convention, is used as the **Meta** key. **Meta** is the standard X window manager key. That is, when you are in a window and want to invoke an X window manager function with a mouse button, you press **Meta** and the mouse button. Some clients, such as **gnuemacs**, also use the **Meta** key extensively. On keyboard 3, any key (except REPEAT) can be a modifier. On keyboard 2, only black keys can be modifiers.

xmodmap Subcommands

The basic **xmodmap** syntax is

> **xmodmap** *filename*

where *filename* is a file of **xmodmap** subcommands that define and undefine modifier keys as follows:

> **add** *modifier–name* = *keysym–name*
> **remove** *modifier–name* = *keysym–name*
> **clear** *modifier–name*
> **keycode** *number* = *keysym–name*

To define modifier keys on Apollo keyboards, the only important subcommands are **add** and **keycode**. (The **clear** subcommand will undefine all keys with that modifier. The **remove** subcommand undoes the assignment on a per–key basis.)

To change the assignments of the default modifiers,

1. Find out the keycodes of the keys that you do want to perform the modifier functions (look in **/usr/lib/X11/keyboard/kb*n*.xprkbd** for keycode assignments).

2. Make your own copy of **/usr/lib/X11/keyboard/xmodmap.kb*n*sample**.

3. Edit in your keycodes and your selected keysym names.

Here are the lines from the **xmodmap.kb2sample** file:

```
!
! Remove any definition for Meta
clear Mod1
!
! "Relabel" key F1 as Control_L
keycode 0x94 = Control_L
! "Relabel" key F2 as Shift_L
keycode 0x95 = Shift_L
! "Relabel" key F3 as Meta_L
keycode 0x96 = Meta_L
!
! Make F1 act as the Control modifier
```

```
add Control = Control_L
! Make F2 act as the Shift modifier
add Shift = Shift_L
! Make F3 act as the Meta modifier
add Mod1 = Meta_L
```

The first command line (**clear Mod1**) tells the server to undefine all keys associated with the **Mod1** modifier. The **keycode** commands indicate that the server is to assign the keysyms for the modifier keys to keycodes 0x94 through 0x96. The first **add** command indicates that the server is to add the control modifier function to the key having the **Control_L** keysym. (The **keycode** commands have already guaranteed that the physical keys have the right keysyms.)

Significant aspects of the **xmodmap** subcommands are the following:

- Case is not significant in a modifier name. "Control" and "CONTROL" are both valid and indicate only one modifier. It is not a keysym, even though it looks like one.
- You *do* need the spaces around the equal sign.
- Case is significant in the keysym name on the right.
- Up to three keys can be the same modifier. (You can have three Control keys, if you wish.)

Scope of Modifier Changes

Modifier definitions outlive the invocation of the X server that is running when they are defined, and indeed they can outlast the login session: they last until they are either redefined with another **xmodmap** or the workstation is rebooted.

Changes you make to the modifiers

- Affect both the DM and X, because they are implemented in the IDM. If you define <F1> as **Control**, it is a control key to the DM as well.
- Cause a key specified as a modifier to lose all its DM keydef meanings—whether the cursor is in X or DM territory. The DM **kd** key definition command does not reflect any changes made with **xmodmap**.

4.2.2 Changing Keysyms with xmodmap

With **xmodmap** you can also change the keycode–to–keysym–text mapping that is generated when the server starts up.

Keysym Subcommand

In our previous example, we assigned modifiers to keycodes, for instance

```
keycode 0x94 = Control_L
```

We could use the **keycode** subcommand to assign any keysym—not just those for modifiers—to a particular keycode. The **keysym** subcommand that would theoretically make the same assignment is

```
keysym F1 = Control_L
```

The **keysym** subcommand works as follows:

1. Take the keysym specified (**F1** in our example).
2. Find the keycode of the key to which **F1** is presently assigned.
3. Assign the keysym **Control_L** to that keycode.

If this **keysym** command is reissued before you log out, then keysym **F1** will no longer be available since we've just replaced it with **Control_L**, and the command will not work. (The **keysym** command is an older **xmodmap** convention.)

Scope of Keysym Definitions

Changes you make to keysyms last the lifetime of the X server. They are therefore shorter lived than modifier changes.

The **xmodmap** commands that affect keysyms ought to be run immediately after the server's "special client" is started (see Section 2.3.4). If the server resets or is killed, rerun the **xmodmap** file to reassign the keysyms. (The modifier changes, however, will last until your workstation is rebooted.)

4.3 Apollo Hardware and Font Considerations

This section describes some of the Apollo-specific considerations involving color, keyboard, and fonts.

4.3.1 Monochrome and Color

The **Xapollo** server allows one visual display, either monochrome or 4- or 8-plane color, depending upon the display hardware. (The number of planes is detected automatically at server startup.)

The 24+ plane true color displays either run by default in 8-plane pseudocolor mode (the only mode which supports X), or they support multiple visuals, in which case X only uses the pseudocolor visual mode.

When running on monochrome DN300, DN330, DN400, DN420, and DN460 workstations, the server uses the portable but slow Monochrome Frame Buffer (MFB) driver code. On all other workstations, the server is optimized to use Apollo native graphics interfaces. (The DM **inv** command has no effect on MFB drivers, but it inverts the color of the X windows on GPR monochrome drivers.)

You can use standard Xlib calls to allocate and free color cells. (We don't support multiple color maps in X in our shared environment.) **Xapollo** calls the Apollo Color Table Manager (CTM) to allocate and free color cells in the color map. It thus does not interfere with other programs(such as the DM) that are executing on the workstation and using the CTM for color management. However, many programs do not use the CTM, and they assume that particular color map cells have certain color values. These programs and the X server will probably mutually interfere.

Also, the X server uses the last two slots of the color map for cursor colors. The DM no longer uses these last two slots, because it now uses just three pairs of colors on a 4-plane system.

4.3.2 Keyboards

In general, Apollo has kept to X's "mechanism not policy" philosophy and has allowed the X server to believe that it is the only window system present and that it has access to all the keys. The user is provided with a server command line option (-K *keyboard.config*) to tailor appropriate access to keys for a shared environment.

Section 4.1.2 provided a thorough description of mapping keycodes to keysyms, partitioning keys between X and the DM, and other related issues. Keysyms for keys not part of the X standard set but found only on Apollo keyboards are kept in **ap_keysym.h**. Writing a client that makes use of these keysyms makes it Apollo-specific and less portable to other hardware platforms.

The server follows the X protocol for Apollo keyboard 3, including international keyboards 3 a–g (appropriate keysyms are automatically assigned from the Latin 1 alphabet to the different international keyboards at server initialization and server reset). The Apollo software (the IDM in particular) emulates "up" transitions for white keys on keyboard 2, so, for that keyboard, mouse modifiers can be defined only on black keys. The IDM thus presents the X server with a nearly uniform keycode set for all keyboards.

(Apollo's keyboards are numbered 2, 3, and 3a–3g. Keyboard 2, as opposed to the newer, low-profile keyboard 3, has no numeric keypad. Keyboards 3a–3g are international keyboards.)

Many options controlled by **xset** (the user preference utility for X) and defined as optional in the X protocol are not appropriate to Apollo keyboard hardware limitations; these include: the pointer motion buffer, threshold, and acceleration factors, key click, bell pitch, and volume. (All of **xset**'s options may be "set," but not all of the settings will actually be effected.)

4.3.3 Fonts

Fonts are distributed in several different formats. The X server can work with the following font formats:

- **snf** (X11 server normal font) format
- **bdf** (character bitmap distribution format) fonts
- Compressed versions of these fonts: **bdf.Z** and **snf.Z**.

The server will uncompress the **.Z** fonts automatically if the BSD4.3 utility **/usr/ucb/uncompress** is available. (You can also run **uncompress** and its companion **/usr/ucb/compress** from the command line.)

The **edfont** utility converts between Apollo's "native **n_bit** font" format and **bdf**, the portable X format. (GPR programs can only work with Apollo **n_bit** fonts.) The **edfont** utility uses a Domain/Dialogue front end and is self-explanatory, but you can type **man edfont** and find out more about it.

The font utilities provided by this version of Domain/X11 replace **fc** with the newer **bdftosnf** font compiler. (Type **man bdftosnf**.) This reads a **bdf** font from the specified file (or from standard input if no file is specified) and writes an **snf** font file to standard output, for example

bdftosnf [*switches*] *font_name*.**bdf** > *font_name*.**snf**

Font Names

Fonts are usually stored as individual files in directories.These files specify font properties, including the names of the fonts. Fonts thus are not named according to the files in which they are stored. Instead, the server reads a font database table out of each directory. This database is created from the font files and specifies the filename for each font. Support for wildcarding and aliasing font names is also available.

Font databases are created by running **mkfontdir** in the directory containing the fonts. **Mkfontdir** takes the font name from the FONT property in each font file and generates a text file, **fonts.dir**, that contains the font names and their associated file names.

Whenever fonts are added to a directory, **mkfontdir** should be rerun so that the server can find the new fonts. If **fonts.dir** is not in a directory, the server will not be able to find any fonts in it. (Type **man mkfontdir** for more information.) In addition, use

```
xset fp rehash
```

after running **mkfontdir**: this will tell the server that you've modified the **fonts.dir** file, and it will reread the font directories.

Wildcard characters can be used in requests for font names to make specifying fonts easier. In addition, font name aliasing is allowed if you store the names of the fonts and their synonyms or aliases in the file **fonts.alias**, which is read before the **fonts.dir** database is scanned by the server. (Each font directory can have both a **fonts.alias** and a **fonts.dir** file.) Each line of this file contains first the alias, then the actual font name to be associated with it, as in the following lines:

```
fixed      6x13
variable   *-helvetica-bold-r-normal-*-*-140-*
```

The R3 font naming follows the conventions used in naming fonts contributed by Adobe Systems, Inc., Bitstream Inc., and Digital Equipment Corporation. The font names contain a lot of information: Foundry, FamilyName, WeightName, Slant, SetwidthName, AddStyleName, PixelSize, PointSize, ResolutionX, ResolutionY, Spacing, etc. Therefore, the font aliased as *variable* above will match Helvetica fonts from any foundry (represented by the first wildcard), any AddStyleName and pixel size (represented by the second and third wildcards), with a point size of 140, and any other characteristics (represented by the fourth wildcard).

The server thus finds fonts by looking in each specified font directory (see the following section on font paths) for a **fonts.alias** file, which it searches for the requested font as an alias. The corresponding font name, if found, is then searched for in the **fonts.dir** file. (If the presumed alias isn't found in **fonts.alias**, it is searched for in **fonts.dir**.) The search continues through the font directories until successful. However, note that the aliases are *cumulative*—in other words, the name being aliased, e.g., 6x13, need *not* be in the same directory as the **fonts.alias** file containing the alias.

R2 Font Names

This release of Domain/X11 uses R3 fonts, but it also includes latent support for R2 fonts. R2 font names are obtained by removing the **.snf** suffix from the file name of a file listed in a font directory. The R2 compatible font directory (**/usr/lib/X11/fonts/oldx11**) and font database in it (**fonts.dir**) are set up with the R2 font names.

Font Paths

The X protocol lets a client tell the server which font path to use. For the X server, this is a list of directory names separated by commas. The default directory names are **/usr/lib/X11/fonts/misc**, **/usr/lib/X11/fonts/75dpi**, and **/usr/lib/X11/fonts/100dpi**.

You can specify a font directory by adding an **-fp** switch to the command line that starts the server, as in

> **-fp** *fontPath*[,*fontPath*$_2$,...,*fontPath*$_n$]

where *fontPath* is the search path for the fonts. The list of directories in which the server looks when trying to open a font is controlled by the font path.

Although most sites may choose to have the server start up with the appropriate font path (using the **-fp** option mentioned above), the path can be overridden by using the **xset** program:

> **xset fp=** *fontPath*[,*fontPath*$_2$,...,*fontPath*$_n$]

The **fp** sets the font path to the directories given in the path argument. (Type **man xset** for more information.) The directories are interpreted by the server, not by the client, and are server-dependent. Directories that do not contain **fonts.dir** databases created by **mkfontdir** will be ignored by the server. Note that font paths that are set via **xset** are reset when the server resets.

To restore the paths that were loaded automatically when the server started or those that were specified in the server's **-fp** command line switch, use

```
xset fp default
```

By default, the X server does not search the R2 **/usr/lib/X11/fonts/oldx11** font directory. You can add this font directory to the **Xapollo** command line with the **-fp** option; or you can use the **xset fp+** command, for example

```
xset fp+  /usr/lib/X11/fonts/oldx11
```

4.4 X Resource Files

Resource files are read by **xrdb**, X's "resource manager." They provide more than just command–line options for clients:

- They let you set options for many clients at once.
- The settings can be changed by command–line options for individual clients.
- The settings may be dynamically specified, based on the particular display being used (for instance).
- The resource files let you give a particular look to your applications, since applications programmed in the same environment (e.g., with the same toolkit) tend to have a common set of options.

4.4.1 Resource File Mechanism

The **xrdb** client, normally run from a **.login** file, reads a user–preference file named **.Xdefaults** in the user's home directory. (User preferences include such aspects as geometry, colors, and fonts.) **xrdb** also lets you load user preferences from any file, and it lets you make run–time, dynamic changes to them. Since the resources are stored in the server, they're available to all clients.

Syntax

The syntax for resource specification provides for setting the attributes of both a class of objects and individual instances of an object. Resources are usually represented, one per line, in the resource file. (See Section 4.4.2 for a sample file.)

For a client–oriented resource specification, the syntax is typically

*client*attribute*: *value*

For a widget–oriented resource specification, the syntax is typically

[*client*]{*,[.*widget*],[.*sub_widget*]}.*attribute*: *value*

The components of this syntax are shown in Table 4-2.

Table 4-2. Syntax Elements for Resource Specifications

Element	Explanation
client	An X client (can be a window manager), *or the class name of an X client (see below).*
attribute	A feature of the last-stated client, widget, or "sub-widget."
value	The actual value to assign to the attribute.
widget	A user interface component, e.g., a scrollbar, command button.
sub_widget	A widget that forms one or more of the components of another widget, as, for example, an **xterm** is comprised of three main widgets: a VT102 window, a Tektronix window, and a menu.
!	Starts a comment line.
\	Ends a line to be continued.
.	The period separates two components that are *immediately* above or below each other in the widget hierarchy (this is called a **tight binding**).
*	A wildcard that matches any number of components. This is called a **loose binding** and is much preferred over tight bindings since *incorrect tight bindings are simply ignored.*

Instance vs. Class

Sometimes various resources in a client or widget are grouped together in the same class. If you look at the man page for **xterm**, for example, you'll see that the individual instances of foreground, cursorColor, and pointerColor resources are all in the class Foreground. This classification allows you to set all of the individual instances of resources in the same class with a single specification, as in

```
        xterm*Foreground: blue        (class)
```

as opposed to setting just the foreground (and not the cursorColor, pointerColor, etc), as in

```
        xterm*foreground: blue        (instance)
```

An instance name always overrides the corresponding class name for that instance. Class names thus allow default values to be specified for all instances of a given type of object, but instance names can be used to specify exceptions. (For the class names of specific clients, run the client, then run **xprop** and click on the client's window.)

Specifying Clients

Client in the specification line, can really be one of two things:

- The class of of the client. By convention, this is the name of the client with the first two letters capitalized.
- The name of a particular client.

For instance, you could use the specification

```
color*Foreground: blue
```

where you just make up the name "color." Then, *any* client's Foreground resources will all be blue, as long as you start the client with the name **color** and it has a class of resources called **Foreground**. For example, you could start an **xterm** with the command

```
xterm -name color
```

Event Translations

Event translations let you redefine an application's reaction to key and button presses (typically). The mappings between events and actions are usually described in the man page for each client. We suggest you read O'Reilly Vol. 3, *X Window System User's Guide*, Chapter 9, and the appropriate parts of Volume 4, *X Toolkit Programming Manual*.

4.4.2 Sample Resource Lines

It's a good idea to look in the **.Xdefaults** files of your local systems so you better understand the "policy" of your environment. This section should be enough to help you read those resource files and modify them for your own purposes. (You may find the defaults files are called **.Xresources**, which is a newer name for the same thing.)

For xclock

For a simple example, consider putting some options for **xclock** into your **~/.Xdefaults** file so that attributes such as the position and update time of your **xclock** are automatic:

```
xclock*mode:                 analog
xclock*borderWidth:          3
xclock*update:               60
xclock*geometry:             40x40-1-50
#ifdef COLOR
xclock*border:               red
xclock*background:           turquoise
xclock*foreground:           black
xclock*hands:                red
xclock*highlight:            black
#endif
```

For xterm

If your **xterms** have fonts that are too small, you may want to take care of this in your **.Xdefaults** file. For example, the following line sets fonts for all your unnamed **xterms** (because it uses the class name for **xterms**—see Section 4.4.5):

```
XTerm.*font: \
-adobe-courier-medium-r-normal--12-120-75-75-m-70-iso8859-1
```

Depending on your **fonts.alias** file, this might be simply

```
XTerm*font: variable_12
```

4.4.3 Finding Resources to Specify

The best places to look for information on which resources may be specified are:

- The man pages for the application
- Any documentation for the widgets used by the application

- The directory **/usr/lib/X11/app-defaults/**for any application resource files

- Any **XtResource** tables in the application or the widget sources

4.4.4 Where to Define the Resources

You can define resources in a number of places. The Resource Manager obtains resource specifications from the following places:

- Any application-specific resource files, usually stored in **/usr/lib/X11/app-defaults/**.

- Any application-specific resource files in the directory named by the environment variable XAPPLRESDIR (default value is $HOME) for programs written using the X Toolkit.

- In **.Xresources** when the **.xinitrc** script for **startx** is used (this **.Xresources** file is merged by **xrdb** into any current resource specification—see Figure 2-3).

- The RESOURCE_MANAGER property on the root window of screen 0; these are stored using **xrdb**. If this property is not defined, then **$HOME/.Xdefaults** will be read to provide compatibility with X10 (although the resource specification format has changed somewhat).

- Any user-specific defaults stored in a file whose name is set in the environment variable XENVIRONMENT.

- The -**xrm** command-line option (for programs written with the X Toolkit).

Resources are usually loaded into the RESOURCE_MANAGER property from a file using the **xrdb** client that is run from whatever script you use to start up X.

4.4.5 Checking a Client's Resources

The easiest way to check the resources for a given client is to use **xprop**. When the cross-hair cursor appears, just click on the client. You'll see output similar to the following:

```
% xprop
WM_COMMAND(STRING) = { "xterm", "-geometry",
      "80x24+0+0", "-name", "login" }
WM_CLIENT_MACHINE(STRING) = "bloom"
WM_CLASS(STRING) = "login", "XTerm"
WM_NORMAL_HINTS(WM_SIZE_HINTS):
      user specified location: 0, 0
      user specified size: 484 by 316
      program specified minimum size:4 by 4
      program specified resize increment: 6 by 13
WM_HINTS(WM_HINTS):
      Client accepts input or input focus: True
      Initial state is Normal State.
WM_ICON_NAME(STRING) = "login"
WM_NAME(STRING) = "login"
```

Chapter 5

Programming with Domain/X11

Contents

Chapter 5

Programming with Domain/X11

This chapter addresses issues unique to writing X clients on Apollo workstations. It explains how to

- Find the include and library files.
- Make existing native graphics applications compatible with the **Xapollo** shared X server.
- Use the FORTRAN and Pascal "interludes" to call Xlib functions from FORTRAN or Pascal.
- Write X clients that perform Apollo native graphics.

For information on writing X clients and programming with Xlib, see the *Xlib Programming Manual* and the *Xlib Reference Manual*. Domain/X11 includes Apollo's version of the standard Version 11 Xlib, as explained in the release document. The Xlib interface on Apollos is completely standard. (See Section 4.3 for a discussion of the X server's implementation of color, fonts, and Apollo keyboards.)

5.1 Compatibility

There are two main compatibility issues that concern the programmer who develops software on Apollo workstations that will run under the Domain/X11 shared X environment:

- Changing existing programs that use the Apollo programming interface and do not use X
- Porting existing programs into Apollo's X environment

We summarize the compatibility issue in the following sections. Section 5.4 discusses in detail the vendor-specific extension to the X protocol that gives access to Apollo native graphics, including using pad calls in an X client and converting a Domain/Dialogue program's graphics to Open Dialogue, which runs in the X environment.

Existing programs that use the Apollo programming interface (DM pad calls, native graphics using GPR, 2D GMR, GSR, and so on) do not have to be modified, recompiled, or relinked in order to run with the shared X server in either DM– or X–root mode. Existing programs that make unsupported RM or IDM calls may not work, as the RM and the IDM have been extensively modified.

However, some existing Apollo graphics programs do not run if invoked directly from an **xterm** X terminal emulator client. These include direct mode, Apollo native programs that don't use the **pad_$create** call but assume the **ios_$stdout** of the window they are being executed from for their own stream ID. They effectively "take over" the DM transcript pad the user runs them from. However, **xterm** clients are not pads. Trying to run such a program in an **xterm** will result in the error message "Attempted operation illegal on this stream (stream manager/IOS)." To start such an application, either move to a DM pad or use the "wrapper" routine **dmwin** (see **man dmwin**). "Frame mode" programs have a similar problem. They should be run from DM pads, not **xterm** windows.

Programs that create their own windows with **pad_$create_window** start properly in either environment. Programmers should consider using **pad_$create_window** to initialize windows in their applications.

5.2 Compiling and Linking an X Client Using C

This section

- Lists the header files that must be included for various Xlib functions
- Explains how to link against the Xlib library
- Provides a brief overview of Apollo's extensions to Xlib

5.2.1 C Header Files for Xlib

Header files for Xlib reside in the directory **/usr/include/X11**. The following is the syntax for including these files:

#include <X11/*name*.h>

The standard header files include the following:

Xlib.h	Header for Xlib. Includes **X.h**. All programs must include at least **Xlib.h**, even if no other header files are included.
keysym.h	Necessary if you will be doing keyboard input using keysyms.
Xresource.h	Necessary if you will use the X resource manager.
Xutil.h	This includes definitions for miscellaneous items such as window manager properties and size hints, and some obscure keysym handling functions.
Xatom.h	Necessary if you will use predefined X properties.

There are also the following Apollo-specific header files:

ap_share.h	This is the C header file necessary if you are going to initialize Apollo native graphics in an X window. (The include file for the share extension is discussed further in Section 5.5.1.)

ap_share.ins.pas	
	Pascal version of **ap_share.h**.
ap_share.ins.ftn	
	FORTRAN version of **ap_share.h**.
ap_keysym.h	Contains keycode/keysym definitions for the keys on Apollo keyboards that are *not* part of the standard X keysym set. Include only if you are writing keyboard-handling routines that make use of Apollo-only keys. Such a client will not be directly portable to other hardware platforms.

The rest of the header files in **/usr/include/X11** are used by the X Toolkit, specific programs, or as part of building the X system and server.

5.2.2 The Xlib Library

Before you actually run Domain/X11 or any of the X client programs, check your **/etc/sys.conf** file to be sure that it lists the following line specifying the Xlib library:

```
lib x11lib, optional
```

This library is a **run-time library** that contains **globally-known** symbols. The symbols are installed in memory when the workstation boots, and then they're available to any process running on your workstation when it references one of them. This allows the X clients to be much smaller, since they are not linked with Xlib. (The word "optional" suppresses the installation warning if the library isn't found and can be omitted.)

On most systems, you need to be root to edit **/etc/sys.conf**.

To keep Domain/X11 compatible with the X Consortium's X11, **libX11.a** is provided as a "dummy" library in **/usr/lib/X11**. When a program is linked against this dummy library, no symbols are resolved, and the runtime library will be used instead. We recommend that you link against the dummy so that programs you write on Apollos will have portable makefiles.

Your BSD compile and link command can be simply

```
cc xprogram_name.c -L/usr/lib/X11 -lX11
```

(The -l in **-lX11** is a small letter "L" to identify the library **libX11.a**. Note that there must be no space between the **-l** and the **X11**.)

If you're in a SysV environment, use a variant of the **cc** command:

```
cc -DSYSV xprogram_name.c -L/usr/lib/X11 -lX11
```

This defines the SYSV symbol and permits the compiler to include the appropriate parts of the Xlib header files.

5.2.3 Apollo Xlib Library Extensions

Apollo's Xlib has two enhancements over the MIT Xlib. First, the library is type-stamped "any" for both runtype and systype. It works correctly with any clients of runtype "any," therefore it will work with code type-stamped "sys5," "sys5.3," "bsd4.2," or "bsd4.3" depending upon which environment the code was compiled under. Second, the library includes the Apollo calls for "share extensions" (**XapolloDisownWindow**, **XapolloGetRectangleForWindow**, etc.) that make it possible to initialize native graphics in an X window.

5.2.4 Other X Libraries

Several X libraries are available with Domain/X11. These include the libraries Xtlib and Xawlib, as well as Xlib.

Xtlib is the "X Toolkit Intrinsics" library. The X toolkit is an object-oriented system in which the "intrinsics" are fundamentally the object-oriented "glue" that handles object construction/destruction, message passing, inheritance, etc. The name for objects layered on this system is "widgets." Xawlib supports the Athena widget set.

These libraries are found in **libXt.a** and **libXaw.a**, so that clients can be linked with them and thus continue to run even if the run-time versions of these libraries change. If you want your clients to use the new features of a new **.a** library, they must be recompiled and relinked. All clients that are -inlib bound must be recompiled and relinked when the libraries change.

5.3 Compiling and Linking in FORTRAN or Pascal

This section explains the provisions for FORTRAN and Pascal programming that Apollo has provided to make it possible to write X clients in either of these languages.

5.3.1 Interludes

Domain/X11 comes with a set of FORTRAN and Pascal interlude subroutines that make it possible to write X clients in FORTRAN and Pascal that call the Xlib library.

Apollo does not supply FORTRAN and Pascal versions of Xlib. Rather, there are FORTRAN and Pascal versions of the five major Xlib include files (**X.h**, **Xlib.h**, **Xresource.h**, **Xatom.h**, and **Xutil.h**), and a set of interlude routines that you can call from FORTRAN or Pascal. They bridge the incompatibilities between these languages' case-insensitivity and call-by-reference parameter passing and C's call-by-value, case-sensitive environment. These interludes in turn call the correct Xlib C routines.

Interludes have been written for every Xlib function call. However, equivalents have not been written for the many C macros found in Xlib clients that return values from the Display structure. These macros are listed in Appendix C of the *Xlib Reference Manual*. You can recognize them in X clients written in C by their mixed-case representation, and because they lack the "X" that precedes all Xlib function calls. For example, **BlackPixel**(*display*, *screen*) is a macro, not a function. Each macro has an equivalent Xlib function call that you can substitute. **BlackPixel**'s equivalent function call is **XBlackPixel**.

The interludes are named exactly the same as the X calls except that the interlude names are in lower case. The Pascal example program **/usr/X11/examples/pascal/xhw1.pas** uses these interludes. (In the example, the case of the names of the interludes doesn't matter because Pascal downcases them.)

FORTRAN and Pascal programmers should note that functions that take string arguments expect the strings to be NULL terminated.

5.3.2 Compiling and Linking

The header files for the interludes are in **/usr/include/X11**. They are called

```
xlib.ins.{pas|ftn}
x.ins.{pas|ftn}
xatom.ins.{pas|ftn}
xutil.ins.{pas|ftn}
xresource.ins.{pas|ftn}
ap_share.ins.{pas|ftn}
```

(Notice that there are no FORTRAN or Pascal versions of the many X Toolkit headers.)

One library, **/lib/x11paslib**, is used for both FORTRAN and Pascal. In a UNIX (BSD or SysV) shell, issue the command

```
ld -o xprog_name xprog_binary  /lib/x11paslib
```

Or, use the Aegis **bind** utility:

```
bind -b xprogram_name xprogram_name.bin /lib/x11paslib
```

The **x11paslib** library is so small that you should link it with your program, but if you wish, you can inlib bind it with the **bind** command

```
bind -b xclient_name xclient_name -inlib "/lib/x11paslib"
```

or issue an **inlib** command before running the client:

```
inlib /lib/x11paslib
```

5.4 Apollo Native Graphics and X Windows

Many Apollo graphics programs depend on being run from a DM window. They do not run in an **xterm**. For these applications, either run them only in DM windows or use the **dmwin** command as explained in Section 5.4.2.

Apollo has added a native graphics extension to X11 in order to let X clients initialize and call Apollo graphics routines in a window. This allows programs with an X interface (whether straight Xlib, Open Dialogue, the MIT X Toolkit, or something else) to surround, say, a 3D GMR, GPR, or PHIGS graphics application.

Such hybrid clients cannot run across a remote client/server connection because Apollo native graphics are not protocol requests. Nor are these clients directly portable to non-Apollo hardware platforms. They do, however, give access to the speed of native graphics; they provide a transition for vendors porting existing native graphics programs to X; and they allow 3D graphics, which won't be available through X until the PEX extension is implemented.

5.4.1 Apollo Windows and Graphics Operations

Most Apollo native window/graphics programs "inherit" the DM transcript session from which they *assume* they were run. (An example of such a program is **/com/dspst**.) They do not try to create new windows and streams, but instead operate on a standard stream (typically **ios_$stdout**). In the user's view, the transcript window is cleared of text, and the graphics session starts up in the same area. Such programs depend on the standard output stream being connected to a DM pad. They cannot be run directly from an **xterm**.

Practically all pad calls take an IOS stream ID as an input argument. This stream is used as the channel over which to send a request to the DM to do something, such as to resize the window in which the pad appears. An Apollo native graphics application that makes pad calls to create its own windows will work fine in an integrated X/DM environment because stream IDs are not used as *input* to the calls **pad_$create_window** and **pad_$create_icon**; instead, these calls create a new window and *return* its stream ID. (An example of such a program is the alarm server.) As a result, programs that initialize with them can be invoked from an SIO line, from a remote node via **/com/crp** or UNIX **rlogin**, as DM background programs via **cpo** or **cps**, in the VT102 terminal emulator, or from a DM transcript.

5.4.2 Standard Stream Pad Calls from xterm

The **dmwin** wrapper program takes as arguments the name of another program and a list of additional arguments to be passed to it (the source is in **/domain_examples/X11/examples/dmwin**, and the binary is in **/usr/bin/X11**). This wrapper program creates a new DM transcript/input pad pair and invokes the given program with the given arguments.

Therefore, instead of typing

> **foo** *arg1 arg2 arg3*

to run the **foo** program from an **xterm**, type

> **dmwin foo** *arg1 arg2 arg3*

Dmwin is written so that you can use redirection for **stderr** and **stdout**, for example

> **dmwin** *~/bin/fancy_graphics* **2>** */tmp/errlog*
>
> **dmwin** *~/bin/just_for_symmetry* **>** *foobar*

Unless a **-f** *filename* switch is used, however, as in

> **dmwin -f** *input_filename ~/bin/process*

standard input is automatically assumed to be from **/dev/tty**. This is done so that users may execute **dmwin** from a script with an "&" at the end of the command line to run it as a background process.

5.5 Apollo Share Extensions

Apollo's extension to the X11 protocol allows X clients to initialize and call Apollo native graphics routines in a window. This mechanism is known as "disowning a window."

By default, X owns only the windows it creates. Disowning a window tells the server not to perform its normal services, presumably because Apollo native graphics are to be displayed in the window.

These services include painting the window border and the window background, and displaying the cursor.

Once a window is disowned, X output to it will be ignored until it becomes owned again. The ignored output is lost. However, you can use X to draw into a disowned window if **Expose** or **VisibilityNotify** events are selected (see Section 5.5.5).

5.5.1 Introduction

Sample programs illustrating an X interface wrapped around GPR programs are in **/domain_examples/X11/examples/x_and_gpr** and **/x_and_gpr_input**. Figure 5-1 shows their output.

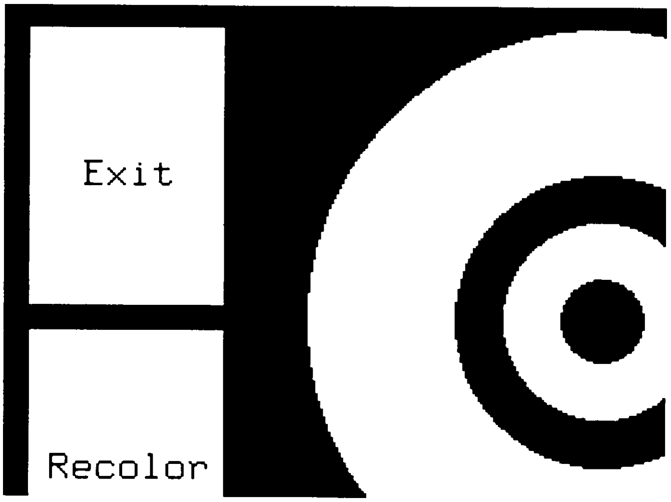


Figure 5-1. x_and_gpr

The share extension consists of five library routines and a new GPR initialization function. The new GPR initialization function described in Section 5.5.2 is resolved in **gprlib**.

The "disowned window" calls are all part of Xlib. You do not need to link to any additional library. The include files needed, in the correct order in which you should include them in your code, are

```
#include <stdio.h>
#include <apollo/base.h>
#include <apollo/gpr.h>
#include <X11/Xlib.h>
#include <X11/ap_share.h>
```

In general, follow these four steps to disown a window and initialize an Apollo native-graphics package:

1. Use **XapolloConnectionIsLocal** to make sure the connection is local. The client and server must run on the same workstation.

2. Use **XapolloGetRectangleForWindow** to obtain the rectangle ID of the window.

3. Use **XapolloDisownWindow** to change ownership of the window from the X Window System to the DM.

4. Use **gpr_$init_rectangle** to initialize GPR ownership of the rectangle/window just disowned and to obtain a pointer to the initial bitmap, to be passed to any subsequent non-GPR initialization function such as for GMR or PHIGS.

On the other side of the coin, when you want X to re-acquire ownership of a window, you can

1. Use **XapolloGetWindowForRectangle** to obtain the window ID of a rectangle owned by the DM, if you don't know the window ID. Although the window is not owned by X and X can't do output to it, you can still make X calls concerning its properties.

2. Use **XapolloOwnRectangle** to acquire ownership of a rectangle for X.

5.5.2 Initializing for Apollo Native Graphics

This section explains the function calls that your program needs to make to disown a window and initialize an Apollo native-graphics program in it.

XapolloConnectionIsLocal

XapolloConnectionIsLocal determines whether the given display connection is local or not. It does not make a server request. It returns **True** if *display* is the local host.

Bool XapolloConnectionIsLocal (*display*)

Display **display;*	A pointer to a **Display** structure.

XapolloGetRectangleForWindow

XapolloGetRectangleForWindow inquires about the mapping between an X window and an RM rectangle; if one exists, it returns the rectangle ID. If the request fails, it returns 0.

gpr_$rect_id_t XapolloGetRectangleForWindow (*display*, *w*, *owns*)

Display **display*;	A pointer to the **Display** structure describing the rectangle.
Window *w;*	The window ID for which the rectangle is being requested.
Bool **owns*;	Returned **True** if X owns the window.

XapolloDisownWindow

XapolloDisownWindow asks the X Window System to relinquish ownership of the rectangle associated with the window *w*. It returns the associated rectangle ID, or it generates an error (and returns 0) if the request fails.

gpr_$rect_id_t XapolloDisownWindow (*display, w*)

Display **display*;	A pointer to the **Display** structure of the window ID.
Window *w*;	The window ID of the window whose ownership is to be transferred to the DM.

gpr_$init_rectangle

The **gpr_$init_rectangle** call gives GPR ownership of the rectangle. It differs from the **gpr_$init** call only in that it initializes on a Rectangle Manager's rectangle ID instead of on a DM pad stream ID. *Always* call **gpr_$init_rectangle** in the process of initializing *any* native graphics in a disowned window. To initialize subsequently one of the Apollo graphics packages other than GPR (such as 2D GMR, 3D , or PHIGS), *also* use their initialization function and pass it the GPR bitmap. The call is declared in **/sys/ins/gpr.ins.{c|pas|ftn}**.

gpr_$init_rectangle (*rect_id, size, hi_plane, bitmap, status*)

gpr_$rect_id_t *rect_id*	The rectangle ID of the window to be initialized.
gpr_$offset_t *size*	The size of the initial rectangle to initialize.
gpr_$rgb_plane_t *hi_plane*	Identifier of the bitmap's highest plane.
gpr_$bitmap_desc_t **bitmap*	Descriptor of the initial bitmap, to be passed to any subsequent non-GPR initialization function.
status_$t *status*	Completion status.

In X, applications always create their own windows, which have window IDs, not stream IDs; the programs manipulate the windows via Xlib calls, and the windows are deleted when the programs exit. Since an X window has no stream ID, use a rectangle ID when you initialize graphics in a disowned X window.

5.5.3 Giving Window Ownership to X

This section explains the two function calls that your program needs to use in order to return ownership of a window to X.

XapolloGetWindowForRectangle

XapolloGetWindowForRectangle inquires about the mapping between an RM rectangle and an X Window. It returns the window ID associated with the specified rectangle and display. If the request fails, it returns **None**.

Window XapolloGetWindowForRectangle (*display*, *r*, *owns*)

Display **display*;	A pointer to the **Display** structure for the rectangle ID.
gpr_$rect_id_t *r*;	The rectangle ID for which the window ID is being requested.
Bool **owns*;	Returned **True** if X owns the window.

XapolloOwnRectangle

XapolloOwnRectangle acquires ownership for the X Window System of the window associated with rectangle *r*. It returns the associated window ID, or it generates an error (and returns **None**) if the request fails.

Window XapolloOwnRectangle (*display*, *r*)

Display **display*;	A pointer to the **Display** structure of the rectangle ID.
gpr_$rect_id_t *r*;	The rectangle ID of the rectangle whose ownership is to be transferred to the X Window System.

5.5.4 Example: Initializing GPR in an X Window

Assume you're given **display** (a display connection) and **w**, an X window.

To initialize GPR in window **w**, you can use the following lines:

```
if(!XapolloConnectionIsLocal(display))
          Error("Not a local connection.");
if(!(rect_id =
    XapolloGetRectangleForWindow(display,w,&owns)))
          Error("GetRectangle failed.");
if(!XapolloDisownWindow(display, w))
          Error("Disown failed.");
gpr_$init_rectangle(rect_id,size,hi_plane,
          bitmap,status);
gpr_$routine;
gpr_$routine;
   ...
```

The recommended way to use GPR in an X window is to let X do the input and GPR do the output. You would have no choice if you were porting a Domain/Dialog application—Domain/Dialog handles input (period). (This means that GPR will draw the cursor even though X is handling input.) The recommended way to use GPR in an X window is to

1. Disown the X window.
2. Select **Expose** events on the window.
3. Initialize GPR in the window.
4. Set **input_ok_if_obs** so you get a cursor.
5. Establish a GPR refresh entry for the window.

Use a main loop (an X event loop) to check for **Expose** events. If an **Expose** event is received for the disowned window, then acquire and release the bitmap so GPR will call the refresh entry point to repaint the window.

5.5.5 X Clients and Disowned Windows

This section discusses what should be correct behavior of X or Open Dialogue clients that are using Apollo native graphics in a "disowned window" with respect to output, exposure events, and waiting for input events. GPR is used as the example, but the discussion applies to other native graphics packages, as well.

Table 5–1 displays the various permutations of states for a window and shows the capabilities of an X client when X owns the window or has selected **Expose** or **VisibilityNotify** events on it.

Table 5–1. Window States as Viewed by X Clients

Window State	**X Output**	**X Input**	**X Cursor**
Owned	yes	yes	yes
Not owned; events are selected	yes	yes	no
Not owned; events not selected	no	yes	no

Output

Basically, X only draws into windows that

- it owns, or that
- **Expose** or **VisibilityNotify** events have been selected on.

An X client may draw into a window as long as either of these conditions is true. If neither is true, the output is ignored.

The only difference between owned and disowned windows is that the X server does not draw the border or background, or render the X cursor, in a disowned window. The purpose of disowning is exactly to inhibit these actions, which would interfere with GPR output.

Exposures

The X server delivers exposure events to any client that has requested them. However, if an application is using GPR in a disowned window, it may have called **gpr_$set_refresh_entry** to establish a refresh entry point that will be called if the window needs refreshing (but only when the program later makes calls to the **gpr_$** functions **acquire_display**, **event_wait**, or **cond_event_wait**). To ensure that a disowned window that has established a refresh entry point gets refreshed whenever necessary, exposure events must be requested for the window, and, when an exposure event is received, the display must be acquired and released.

If you use the **gpr_$set_obscured_opt** function, you'll want to use the enum constant **gpr_$input_ok_if_obs** as the first (i.e., **if_obscured**) argument to that call. This value specifies that the GPR window will accept input and cursor display commands even when the window is obscured. (If you've used **gpr_$ok_if_obs** in the past, you'll probably want to use **gpr_$input_ok_if_obs** instead.)

Input

In general, input events will be permanently routed to the first facility that enables input on a window, whether GPR or X. Input "ownership" cannot be passed back and forth between the two facilities, as output ownership can. However, if a window has not had input enabled on it by anyone as yet, the input event is passed up the hierarchy to the window's parent, and so on, until it reaches a window which has enabled input on it, or until it gets to the root. If the root owner is the DM, the DM gets the event; if the root owner is X, the event is routed to any client that has enabled that event type on the root—if there is no such client, the event will be discarded.

Lifetime of a Window

A window's lifetime is affected by who created it.

Windows created by X are created at the request of a particular X client. They live only as long as the client's connection is in effect, unless the client takes special action to allow them to be preserved.

A window created by someone else and acquired by an X client will remain until its original owner deletes it, even if the client's connection goes away. However, as long as X owns or has selected **Expose** or **VisibilityNotify** events on a window, the underlying rectangle will remain even if its original creator deletes it.

Keyboard Events and GPR

The Low-Profile, Model II keyboard actually transmits down/up events for each key. The Low-Profile, Model I keyboard does not generate down/up events for each key. For this Model I keyboard, pseudo up/down events are generated by the system; for example, if the keyboard transmits "A," the following events are generated:

- left-shift-down
- a-key-down
- a-key-up
- left-shift-key-up

Thus, on the Model I keyboards, receipt of **gpr_$physical_keys** from the modifier keys (shift, control, etc.) do not necessarily correspond to actual state changes of these keys.

When an application runs under the Display Manager, keyboard events are tied to the mouse cursor. When an application receives a **gpr_$left_window** event, it cannot receive keyboard input. If the application is running under an X window manager, however, it is possible for the keyboard focus and the mouse focus to be in different windows. (The focus determines which window will receive input.) To find out when the keyboard focus has entered and left the window, an application can enable **gpr_$kbd_entered_window** and **gpr_$kbd_left_window**. These events are similar to the events **gpr_$entered_window** and **gpr_$left_window**, which occur when the mouse focus enters and leaves the window. An event of type **gpr_$kbd_entered_window** occurs when the keyboard focus enters the window; **gpr_$kbd_left_window** occurs when the keyboard focus leaves the window.

If you are running under an X window manager and you have not enabled **gpr_$kbd_entered_window**, then **gpr_$entered_window** means that either the locator (mouse) or the keyboard has entered the window. Only one event is generated even though both may enter the window. The event is generated when the first mouse locator or keyboard enters.

Similarly, if you are running under an X window manager and you have not enabled **gpr_$kbd_left_window**, **gpr_$left_window** means that the locator (mouse) or keyboard has left the window. Only one event is generated even though both may leave the window. The event is generated when the last mouse locator or keyboard leaves.

Open Dialogue and GPR

Consider an application written to use Open Dialogue for its user interface with an application program that wishes to use GPR (or GMR, GSR, etc.) to do graphics in a window. In this case, Open Dialogue creates the top-level window, draws Open Dialogue graphic objects within it, and creates the window for the application's use. Thus the GPR window is originally an X window. The application then causes X to disown the GPR window in which application code will initialize GPR. (Figure 5-2 illustrates the **x_and_opd** example which we have provided in the past, but which you may now find in the **/examples** directory for Release 2 of Open Dialogue.)

For output, GPR is then used in the window. Output is bracketed by calls to **gpr_$acquire_display** and **gpr_$release_display**.

Open Dialogue manages all the input for an application, so the application should not attempt to initialize GPR input in the window. The application gets all events relating to its GPR window through the use of Open Dialogue's **dlg_x_event** function, in the form of X events. These events include input events and exposures. The application may respond to the input events by doing output. It may choose to respond to exposure events simply by redrawing the affected area of the GPR window directly. If the application chooses to operate in this way, it is using Open Dialogue as if it were doing X graphics instead of GPR graphics in the window. On the other hand, the program may have established a routine to be called by GPR via **gpr_$set_refresh_entry**. In this case, the program should respond to exposures simply by acquiring and releasing the display: the acquire will cause the refresh entry point to be called as a side effect.

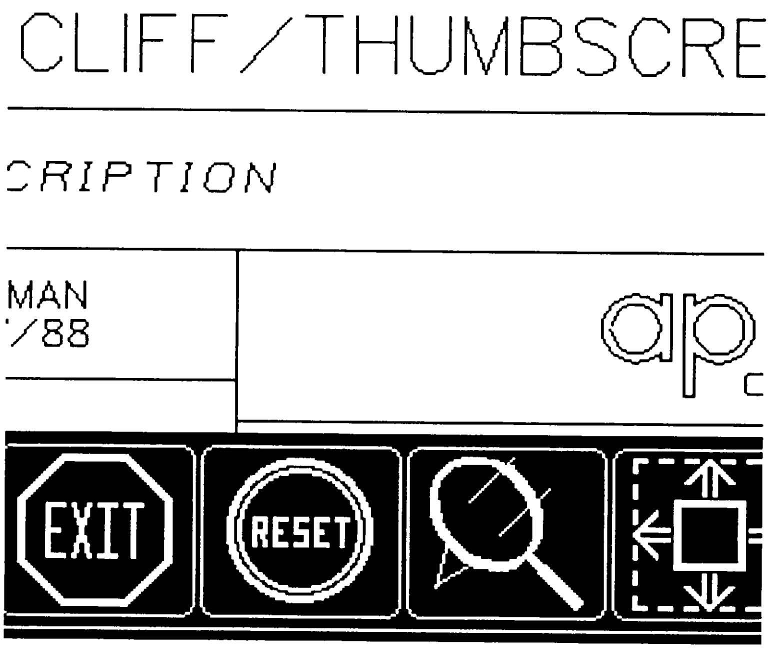


Figure 5–2. x_and_opd

Using X and GPR in Separate Windows

A program may wish to be an X client and a GPR user simultaneously but independently. It may have created some X windows in which it does X graphics, and some Apollo native windows in which it uses GPR. In this case, there need be no interaction between the two aspects of the program. Any GPR functions are legal on the Apollo windows, and any X functions are legal on the X windows. One particular case of this situation would be a program that uses X via Open Dialogue, but creates separate, top–level Apollo windows to draw in as well.

Do *not* acquire the display when calling X over a local connection or you'll be in a deadlock with the X server. No Xlib calls, even if they do no output, should be made between acquiring and releasing the display.

Appendixes

Appendix A Checklist for Xapollo and Clients

Appendix B Troubleshooting Hints

Appendix A

Checklist for Xapollo and Clients

This checklist provides guidelines to help you get set up correctly. See Appendix B for troubleshooting hints.

A.1 TCP/IP

- ☐ **/etc/daemons** contains zero–length files **tcpd** and **routed** (see Section C.1).
- ☐ **/etc/hosts** registers local host (see Section 1.3.2).
- ☐ **/etc/ping** *hostname* is successful (see Section C.3).

A.2 Run–Time Library

- ☐ **/etc/sys.conf** lists **x11lib** (see Section 5.2.2).
- ☐ Reboot after adding to **/etc/sys.conf** (see Section 2.1).

A.3 Environment Variables

- ☐ PATH includes **/usr/bin/X11** (see Section 3.1.4).

- ☐ DISPLAY specifies *hostname*:**0** or **:0** (see Section 3.1.3).

A.4 X Server

- ☐ At least 16 pseudo-ttys are available (see Section 2.1).
- ☐ Share mode and root ownership have been specified by **s** and **r** flags in **Xapollo** command (see Sections 2.3.2 and 2.3.3).
- ☐ A window manager has been selected, e.g., by issuing **wmgr -off, uwm &** (see Sections 2.3.2, 2.3.3, and 2.5).
- ☐ If you run **Xapollo** automatically at boot time, you use **/etc/rc** with file **/etc/daemons/Xapollo** (see Section 2.2).
- ☐ If you run **Xapollo** at login, you use **startx** (see Section 2.2.4) or a script of your own that includes **xinit** (2.3.4).
- ☐ If you run **Xapollo** manually, you run it in the background via **startx** with **&** (see Section 2.3).
- ☐ **Xapollo** is *not* run manually via **Xapollo &** (see Section 1.1.3).
- ☐ Reset is prevented with **xinit**, **xclock**, or a DM connection (see Section 2.3.4).

A.5 X Clients

- ☐ Set **-display** or use DISPLAY variable (see Section 3.1.3).
- ☐ Run in background with **&** as needed (see Section 3.2.1).
- ☐ Use **dmwin** when executing DM-specific graphics applications from an **xterm** (see Sections 3.3.8, 5.1, and 5.4.2).
- ☐ Workstation for remote display has TCP and **Xapollo** running, and your node listed in its **/etc/X0.hosts** file (see Section 3.2.4).

Appendix B

Troubleshooting Hints

This appendix provides some additional notes to help you solve some of the problems that you may find if you are new to X.

B.1 X Server Initializes Slowly

The server can take up to 60 seconds to initialize fully. The length of time is related to several factors:

- If you don't run TCP, your **Xapollo** delays startup for about 50 seconds while it tries to make a TCP connection before it prints an error message and goes on to UDS setup.
- The number of hosts listed in the **/etc/X0.hosts** file for which the server must retrieve real network addresses. The more entries, the longer it takes.
- CPU speed, network traffic, and system load.
- Whether the DM failed to start.

B.2 Background, Rubberbanding, Cursor Anomalies

If your background window has any of the following problems

- Two cursors
- An erratically moving (**warping**) cursor
- Improper repainting
- Missing rubberband lines drawn by your window manager

check that you're only running one window manager. If X owns the root, use **wmgr -off** and run **uwm** or another X window manager. If the DM owns the root, issue the **xownroot -off** command.

If you switch from the DM owning the root to X owning the root, take the following steps:

1. Use **wmgr -off**.
2. Issue the **xownroot -on** command.
3. Start **uwm** or another X window manger.

If you switch from the X owning the root to DM owning the root, the following steps:

1. Kill your X window manager.
2. Issue the **xownroot -off** command.
3. Use **wmgr -on**.

B.3 Clients Can't Open Display

Every time an X client program runs, it must be told which server to connect to. All clients accept the command-line option **-display**; if this option isn't found, they check the DISPLAY environment variable. If you don't issue the **-display** option or set the DISPLAY variable correctly, any X program that you try to run will fail with some variant of the "Can't Open Display" error message. See Section 3.1.3.

B.4 Clients Initialize Slowly

Some clients, such as **xcalc**, contain many windows. The number of windows affects the initialization time.

B.5 Color Slots

The DM program **lcm** and DSEE's "show derivation" command do not comply with the Color Table Manager.

B.6 Cursor Remains a White Square

If the background appears, but the cursor is a white square that doesn't change, make sure that the fonts have been installed (in particular, the font named **cursor.snf** in the directory **/usr/lib/X11/fonts/misc/**; see the configuration parameter DefaultFontPath). Also make sure that there is a file named **fonts.dir** in each font directory. This file is created by the **mkfontdir** program and is used by the server to find fonts in a directory. (See Section 4.3.3 for more information on font paths.)

B.7 Diskless Nodes

You can start **Xapollo** and run X clients on diskless nodes. Make sure you have TCP running on the diskless node and that **ping** echoes the diskless node.

B.8 Error Messages Are Confusing

The server was—for the most part—written at MIT, not Apollo: this means we didn't write all the error messages. Some of them are confusing, but try a sense of humor—occasionally they're actually funny:

```
Fatal server bug!
RC HORRIBLE ERROR...
MaskForEvent: bogus event number
Somebody is setting NullCursor
Impossible keyboard event
Freeing resource id=%X which isn't there
```

B.9 Fonts Aren't Found

X clients programmed for Release 2 may not find the fonts they need by default. If you want to use Release 2 fonts, add the **oldx11** font directory to the command line that starts the server or to an **xset** invocation. (See Section 4.3.3 for an example.) Any time you run **mkfontdir** to update the **fonts.dir** file in a directory, use **xset fp rehash** to tell the server to read that file.

B.10 Host Names

For *hostname* in your DISPLAY variable or **-display** switch, use the TCP/IP hostname of your workstation (usually just the name of your workstation without the // prefix). Use the **hostname** command to verify it.

B.11 Illegal Stream Operations

Some existing programs do not run if invoked directly from an **xterm** X terminal emulator client. These include direct mode, Apollo native programs that *don't* use the **pad_$create** call but assume the **ios_$stdout** of the window they are being executed from for their own stream ID. They effectively "take over" the DM transcript pad from which the user runs them. However, **xterm** clients are not pads and have no **ios_$stdout** associated with them. Trying to run such a program in an **xterm** will result in the error message "Attempted operation illegal on this stream (stream manager/IOS)." To start such an application, either move to a DM pad or use the "wrapper" routine **dmwin** (see **man dmwin**). "Frame mode" programs have a similar problem. They should be run from DM pads, not **xterm** windows.

B.12 Insufficient Pseudo Terminals

Have at least 16 "pseudo-ttys" in /**dev**. If the installation process did not create any or enough of these, ask your system or network administrator to issue the appropriate commands. See Section 2.1.

B.13 Interleaf and uwm Key Questions

If you have the following two lines in your **.uwmrc** file, the left and middle mouse buttons won't work in Interleaf.

```
f.iconify =    :i  :        left down
f.move =       :i  :        middle down
```

You've told **uwm** to grab two mouse buttons. Unlike the DM, which uses bottom up routing, **uwm** tends to be anti-social and just grab what it needs. Use shifted mouse buttons instead.

B.14 Key Mappings and Definitions

If **xmodmap** defines modifier keys in X that are needed by a DM application program, the application program will have to use other keys, or the modifier keys will have to be reassigned. It's better to reassign them in **xmodmap** than to exclude them from the server with the **-K** switch on the **Xapollo** command.

Keys previously used for control (for example, **Upbox** and **Downbox**) in earlier servers can now be modified by using **xmodmap**. Incorporate the following line in your startup file for your login:

```
xmodmap ~/.keymap.km
```

The *~/.keymap.km* file consists of modifiers for keys, as in this example:

```
! On an Apollo, the following keys have the codes:
!
! 0x95   0xff08    BackSpace
! 0x7f   0xffff    Delete
! 0xff   0xffe1    Shift_L
! 0xfe   0xffe3    Control_L

clear Control
clear Shift
clear Mod1

! You must always have 0xfe and 0xff defined as:
keycode 0xff = Shift_L
keycode 0xfe = Control_L
add Control = Control_L
add Shift = Shift_L

! Make the delete key the same as the backspace key
! keycode 0x7f = BackSpace
! Add shift and control keys for our mouse.

! These are the 'normal' definitions;
| Begin = Control_R, End = Shift_R, F21 = Meta_R

keycode 0x8d = Control_R
keycode 0x8f = Shift_R
keycode 0x90 = Meta_R
```

B.15 Rebooting Needed

Reboot after updating your /**etc/sys.conf** file. See Section 5.2.2 for more information.

Reboot after you add files to your node's **/etc/daemons** directory, or after editing files in your **/etc** directory. See Section 2.2.2.

B.16 Root Operations

You need to log in as root or use the **su** command (see **man su**) to perform the following:

- Kill **Xapollo** started by **/etc/rc**.
- Create pseudo tty's in **/dev**.
- Edit **/etc/sys.conf** (if necessary).
- Edit **/etc/rc** to modify **Xapollo** flags, etc.
- Execute **/etc/sys_sh** to run **/etc/rc**.
- Kill **xterms**.
- Edit **/etc/X0.hosts**.
- Edit **/etc/hosts**.

B.17 TCP Problems

Occasionally, you might get a "can't open display" type of error message if you set your DISPLAY variable to anything other than :0. This can mean that something is wrong with TCP. Does your node have correct links for **/etc/hosts**, **/etc/hosts.equiv**, **/etc/networks**? Do you have a correct **/etc/services** file? Are you running **tcpd** and **routed** (and also **inetd**, if you want **rsh**, **rlogin**, etc.)? Check the TCP manual for further information.

B.18 Unpacking Files

If your administrator hasn't unpacked or uncompressed the document files, you have to issue the following commands (which are explained more in the **/usr/src/X11/README** file) as appropriate:

```
/usr/ucb/uncompress  filename.tar.Z
/bin/tar xvf  filename.tar
```

(Files ending in **.Z** are Ziv–Lempel compressed files.)

Appendixes

B.19 Using a BitPad (Digitizer)

Apollo supports the Summagraphics Bitpad 1 data tablet, which can be used instead of a mouse to control input. Application programs can't tell the difference between mouse input and tablet input, since they all look like locator data. As a matter of fact that's why it's called "locator data" and not "mouse data."

The tablet is enabled by editing **/sys/node_data/etc/rc.user** to uncomment the following lines:

```
#  Summagraphic bitpad support.
 if [ -f /sys/dm/spbl ]; then
         (echo " bitpad\c" >/dev/console)
         /sys/dm/sbpl /dev/sio2 L &
 fi
```

B.20 Using <REPEAT> for Meta

If you give the "incantation" to **xmodmap** to set up the REPEAT key as **Meta**, the key will work as **Meta**, but it also repeats. You can avoid getting three or four repeated characters per "metafied" key by using the command

```
/com/kbm -R none
```

See **help** or **man** pages for **kbm**.

B.21 Vertical (Rotated) Fonts

There is a way to draw vertical text in X to label the vertical axis of a graph, but it isn't easy. There's an R2 font called **rot-s16** which has characters oriented going down the page. If you image each character with a separate Xlib call, each on its own baseline, you can label axes.

B.22 XIO: Operation Would Block

If you get the message

```
XIO: Operation would block
```

from an X client program you're developing, it can mean that you've made the mistake of defining the variable **errno** somewhere in one of your C modules. Wherever you declare **errno**, you should use the following declaration:

```
extern int errno;
```

If you declare **errno** without declaring it **extern**, then your program may refer to a different location than the various shared libraries. One consequence of this is that Xlib's internal code can pick up incorrect data.

Appendix C

Checking TCP Out

This appendix provides some hints on making sure that TCP is working correctly on your workstation.

C.1 Connections: Starting the TCP/IP Daemon

The TCP/IP daemon (server) should already be fully initialized and running before you start the **Xapollo** server, otherwise **Xapollo** will quit trying to make a TCP/IP connection and will come up after about 60 seconds, but it will use a different form of interprocess communication (UDS). If you don't use TCP, only local connections are possible—i.e., the server and client have to be on the same workstation. Start TCP by using the following steps:

1. Create files named **tcpd** and **routed** in your **/etc/daemons** directory by issuing the commands

   ```
   touch /etc/daemons/tcpd
   touch /etc/daemons/routed
   ```

   (If your workstation is not operating in a network, you don't need the file **routed**.)

2. Make sure that your **/etc/hosts** file contains an entry with your node name. As mentioned in Section 1.3.2, this may be a task for your system or network administrator.

3. Make sure that your **/etc/rc.local** file script exists. You probably don't have to edit it unless you're using more than one network interface, but the standard file *must* be there. (Read your copy of the file if you're concerned about editing it, and ask someone who has root privileges to edit it, if you don't have permission.)

If you follow these steps, TCP will start up when your node starts up, because the **rc.local** file is called from **/etc/rc** during node startup.

C.2 Making Sure that TCP Is Running

You can check for the presence of TCP's **tcpd** process by issuing the UNIX command **ps -ax** or the Aegis command **pst**. Check for **/etc/tcpd** and **/etc/routed**. See Figure C-1 for an example.

```
PID TTY      STAT  TIME COMMAND
  1 ?        S <   0:39 /etc/init
  2 ?        R  2986:43 null
          .
          .
          .
 85 ?        S    14:12 /etc/tcpd
 90 ?        S     2:12 /etc/routed -h -f
```

Figure C-1. The Process List with **tcpd**

C.3 Using ping to Check TCP

The TCP server can be running, but it may not be configured correctly. This is when you use **ping**.

The **ping** utility was developed to help test the DARPA Internet. It issues a request for an echo from a host or gateway. For our purposes, you can run it with the command

```
/etc/ping hostname 56 4
```

where *hostname* is the name of your node, **56** is the packet size used by Apollo's TCP, and **4** is the number of echoes requested. (Note that **ping** has to be **setuid root** for it to work—see **man setroot**.) First run **ping** on the local host, to verify that the local network interface is up and running. Then, use **ping** on hosts and gateways further and further away, as needed.

Ping prints one line of output for every echo returned. Figure C-2 shows the output for a node whose TCP/IP is working. Figure C-3 shows the response for a node that is properly listed in **/etc/hosts**, as seen in the line in that file that shows

```
    192.9.3.21        naughty_node
```

but, evidently, the node has some other problem. (For our example, our naming server was unavailable when we booted **//naughty_node**. Try to reboot your workstation and see if that helps.)

```
bsd4.3 101 % /etc/ping good_node 56 4
PING good_node: 56 data bytes
64 bytes from 192.9.3.118: icmp_seq=0. time=532. ms
64 bytes from 192.9.3.118: icmp_seq=1. time=104. ms
64 bytes from 192.9.3.118: icmp_seq=2. time=21. ms
64 bytes from 192.9.3.118: icmp_seq=2. time=11. ms
----good_node PING Statistics----
4 packets transmitted, 4 packets received,
0% packet loss
round-trip (ms)  min/avg/max = 10/98/532
```

Figure C-2. A Good **ping**

Ping is intended for use in network testing, measurement, and management. It should be used primarily for manual fault isolation. Because of the load it could impose on the network, don't use it during normal operations or from automated scripts, especially without specifying the number of echoes requested. (Because **ping** can be stopped with the **Quit** key, which can differ from user to user, we suggest you automatically limit the number of echoes.)

```
bsd4.3 133 % /etc/ping naughty_node 56 4
PING naughty_node: 56 data bytes
sendto: Network is unreachable
ping: wrote naughty_node 64 chars, ret=-1
sendto: Network is unreachable

----naughty_node PING Statistics----
2 packets transmitted, 0 packets received,
100% packet loss
```

Figure C–3. A Bad **ping**

Appendix D

Sample GPR Program

This appendix lists a sample GPR program that runs in a "disowned" X window. The program implements the recommendations in Chapter 5 and is in file **/usr/X11/examples/x_and_gpr/x_share_gpr.c**.

```
/*********************

  x_share_gpr.c


        07/12/89    bre    Use /usr/include/apollo .h's
        04/07/89    bow    Use unmap gravity
        03/27/89    bre    Remove xgpr.ins.c
        10/17/88    wol    Use AllocNamedColor
        09/88       pcb    created


  This simple program has an iconic interface to a graphics
window that displays a set of concentric circles.  The colors
of the circles can be changed by clicking on the "recolor"
icon.  The program is terminated by the "exit" icon.

  The program shows how to run GPR within an X window.Although
this is a rather simplistic example, it illustrates several
key features.In particular, it demonstrates methods for:

      - Starting an X client, including connecting to a server,
        opening windows, and initializing input
        - Allocating colors in the default color map
        - Initializing GPR to run within an X window
        - Implementing an X event processing loop
        - Scaling and redrawing both X and GPR windows
        - Terminating GPR and the X client

  There are several areas in which X and GPR overlap in terms
of capabilities.  Input and color table management are two
such areas. In this example, the X interface is used for both.

  This example rescales itself whenever the main window is
resized and redraws exposed regions whenever any of the win-
dows are unobscured. This is not a default feature of X.  It
was implemented using standard X mechanisms for the X windows
and standard GPR mechanisms for the GPR window. Even though
GPR mechanisms are used to refresh the GPR window, note that X
expose events are used to detect exposure in the GPR window.

  X, by default, allows updating of partially obscured win-
dows, while GPR does not.  During initialization of the GPR
window, drawing in obscured windows is explicitly enabled.


 A WORD OF CAUTION: do not make any X calls between a gpr_$ac-
quire_display and a   gpr_$release_display.  EVERY X call at-
tempts to acquire the display, and making one during this pe-
riod will essentially hang your program.

**********/


/*  Standard system and GPR include files */

#include <stdio.h>

/* NOTE:  gpr.ins.c must be included before the Xlib.h header
  file. Xlib.h includes <sys/types.h> which defines the mac-
  ros "major" and "minor".  This causes a problem with the
  gpr.ins.c definition of

        typedef struct {
            short major,minor;
        } gpr_$version_t;
*/

#include <apollo/base.h>
#include <apollo/gpr.h>
#include <apollo/error.h>


/*  Standard X include file */

#include <X11/Xlib.h>

/*  Apollo Share extension include files */

#include <X11/ap_share.h>
```

```
/*  Constants used for drawing text in the icons */

#define ICON_FONT ("9x15")
#define ICON_FONT_CHAR_WIDTH  9
#define ICON_FONT_CHAR_HEIGHT 15

#define EXIT_STRING                ("Exit")
#define EXIT_STRING_WIDTH          (ICON_FONT_CHAR_WIDTH *
sizeof(EXIT_STRING))
#define EXIT_STRING_HEIGHT         (ICON_FONT_CHAR_HEIGHT)

#define RECOLOR_STRING             ("Recolor")
#define RECOLOR_STRING_WIDTH       (ICON_FONT_CHAR_WIDTH *
sizeof(RECOLOR_STRING))
#define RECOLOR_STRING_HEIGHT      (ICON_FONT_CHAR_HEIGHT >> 1)


/* Constants and variables used for loading the color table */

#define BLACK                     0
#define RED                       1
#define GREEN                     2
#define BLUE                      3
#define CYAN                      4
#define YELLOW                    5
#define MAGENTA                   6
#define WHITE                     7
#define NUM_COLORS                (WHITE + 1)

char *clr_name[] = {"black", "red", "green", "blue", "cyan",
                    "yellow","magenta", "white"};

/* Constants used for hiliting and unhiliting the icons */

#define ICON_HILITE             BLUE
#define ICON_BG                 WHITE
#define ICON_TEXT               BLACK


/*  Global variable declarations */

Display  *display;     /* X display structure */
Window   main_window;  /* X window resource id of
                          background window */
Window   exit_window;  /* X window id of exit icon window */
Window   recolor_window;   /* X window id of recolor icon
                              window */
Window   apollo_window;    /* X window id of Apollo graphics
                              window */
unsigned long colors [2 * NUM_COLORS]; /* array of color
                                          table indices */
GC       icon_gc;              /* graphic context used for X
                                  drawing */
unsigned int width, height;  /* current size of main window */
int     hilite;                /* True if hilite supported */
gpr_$bitmap_desc_t bitmap;     /* graphics window bitmap ID */
void refresh_procedure ();     /* GPR refresh entry routine */
```

```
/* main gets the application up and running, processes events,
and terminates when the user is finished. */


main (argc, argv)

int     argc;
char    *argv [];

{

    /* Standard X initialization: connect to an X server,
       create windows, and enable events. */
    start_x ();

    /* Set up for running Apollo graphics in the X window */

    start_gpr_in_x_window (apollo_window);

    /* Map the windows */

    XMapSubwindows (display, main_window);
    XMapWindow (display, main_window);

    /* Now that everything is set up, process events until
       the user wants to terminate */

    process_x_events ();

    /* Termination: standard calls for both gpr and X */

    terminate_gpr ();
    terminate_x (NULL);

}

/* start_x performs standard X initialization:  it connects to
   an X server, allocates and initializes the color table
   entries that will be used, creates windows, enable events,
   and make the windows displayable */

start_x ()

{
int   screen;      /* default screen identifier */
Colormap cmap;     /* default colormap identifier */
short main_color = BLACK; /* main window background color */
short icon_color = ICON_BG;/* icon window background color */
short apollo_color = BLACK;/* Apollo graphics window
                              background color */
XColor clr_cell[NUM_COLORS], rgb_def[NUM_COLORS];
XSetWindowAttributes attrib;
unsigned long   valuemask;
int   i;
```

```
    /* Connect to an X server */

    if (!(display = XOpenDisplay (NULL))) {
        fprintf (stderr, "XOpenDisplay failed\n");
        exit (1);
    }

    screen = DefaultScreen (display);

    /*  Set up the colormap.  If the exact color is
       not already defined then the Xserver will allocate
       the closest color available.  This method of defining
       the CLT will work on all Displays. */

    cmap = DefaultColormap (display, screen);

    for (i=0; i<NUM_COLORS; i++) {
         XAllocNamedColor(display, cmap, clr_name[i],
                             &clr_cell[i], &rgb_def[i]);
         colors[i] = clr_cell[i].pixel;
    }

    /* Replicate the color indices in the colors array to
       make the drawing loop faster. */

    for (i=0; i<NUM_COLORS; i++)
        colors[i+NUM_COLORS] = colors[i];

    /* Determine if the Display device has the hilite color.
*/

    hilite = ((clr_cell[ICON_HILITE].pixel
                 != clr_cell[ICON_BG].pixel)
              && (clr_cell[ICON_HILITE].pixel
                 != clr_cell[ICON_TEXT].pixel));

    /* Create a main window 1/4 screen size and located in the
       center of the screen */

    width = DisplayWidth (display, screen) >> 1;
    height = DisplayHeight (display, screen) >> 1;

    main_window = XCreateSimpleWindow (display,
                    RootWindow(display,DefaultScreen(display)),
                    width >> 1, height >> 1, width, height, 0,
                    colors [main_color], colors [main_color]);

    /* Create the icon windows */

    exit_window=XCreateSimpleWindow(display,main_window,10,10,
                    (width >> 2) - 20, (height >> 1) - 15, 0,
                    colors [icon_color],colors [icon_color]);

    recolor_window=XCreateSimpleWindow(display,main_window,10,
                    (height >> 1) + 5,
                    (width >> 2) -20, (height >> 1)-15, 0,
                    colors [icon_color],colors [icon_color]);
        /* Create a window for the Apollo graphics */

        apollo_window = XCreateSimpleWindow (display, main_window,
                             (width >> 2), 10,
                             (width >> 2) * 3 - 10, height - 20, 0,
                             colors [apollo_color],
                             colors [apollo_color]);

        /* Select UnmapGravity on children */

        attrib.win_gravity = UnmapGravity;
        valuemask = CWWinGravity;
        XChangeWindowAttributes(display,exit_window,valuemask,
                                  &attrib);
        XChangeWindowAttributes(display,recolor_window,valuemask,
                                  &attrib);
        XChangeWindowAttributes(display,apollo_window,valuemask,
                                  &attrib);

        /* Select events -- BEFORE mapping the windows! */

        XSelectInput (display, main_window,
                      ExposureMask | StructureNotifyMask);
        XSelectInput (display, exit_window,
                      ExposureMask | StructureNotifyMask
                          | ButtonPressMask | EnterWindowMask
                          | LeaveWindowMask);
        XSelectInput (display, recolor_window,
                      ExposureMask | StructureNotifyMask
                          | ButtonPressMask | EnterWindowMask
                          | LeaveWindowMask);
        XSelectInput (display, apollo_window,
                      ExposureMask | StructureNotifyMask);

        /* Initialize icon GC */

        init_icon_GC ();

    }

    /* init_icon_GC creates the graphic context that will be used
             for drawing text in the icon windows */

    init_icon_GC ()

    {
    unsigned long  gc_mask = 0;    /* mask of graphic context
                                      fields to set */
    XGCValues      gc_values;      /* values to assign to graphic
                                      context fields */

        gc_values.function = GXcopy;
        gc_values.plane_mask = AllPlanes;
        gc_values.font = XLoadFont (display, ICON_FONT);
        gc_values.foreground = colors [ICON_TEXT];
        gc_mask |= GCFunction|GCPlaneMask|GCFont|GCForeground;
    icon_gc=XCreateGC(display,main_window,gc_mask,&gc_values);
}

/* start_gpr_in_x_window removes an X window from X's sphere
   of control, starts GPR in that window, and establishes the
   appropriate GPR environment for the window. */

start_gpr_in_x_window (window)

Window  window;             /* X window resource id */
{
gpr_$rect_id_t  rect_id;   /* rectangle id of the window */
Bool            owns;      /* true if X owns the window */
gpr_$offset_t   size;
gpr_$rgb_plane_t hi_plane;
status_$t       status;    /* gpr call return status */

     /* Make sure the connection is local, get the rectangle id
        that corresponds to the Apollo window, and tell X to
        disown the window */

     if (!XapolloConnectionIsLocal (display))
             terminate_x ("Not a local connection.");
     if (!(rect_id=XapolloGetRectangleForWindow(display,window,
                                                        &owns)))
             terminate_x ("GetRectangle failed.");
     if (!XapolloDisownWindow (display, window))
             terminate_x ("Disown failed.");

     hi_plane=DisplayPlanes(display,DefaultScreen(display))-1;

     /* Make the initial bitmap full-screen sized so the
        window will look right if made larger */

     size.x_size=(short)DisplayWidth(display,
                                     DefaultScreen (display));
     size.y_size=(short)DisplayHeight(display,
                                      DefaultScreen (display));

     /* Initialize GPR using the rectangle id */

     gpr_$init_rectangle(rect_id,size,hi_plane,&bitmap,
                                                     &status);
     check ("gpr_$init_rectangle failed.", status);

     /* Enable cursor echo within the window */

     gpr_$set_cursor_active (true, &status);
     check ("gpr_$set_cursor_active failed.", status);

     /* Set up for refreshing the graphics after the window
        state changes */

     gpr_$set_obscured_opt (gpr_$input_ok_if_obs, &status);
     check ("gpr_$set_obscured_opt failed.", status);
     gpr_$set_refresh_entry(refresh_procedure,
                            (gpr_$rhdm_pr_t) NULL,&status);
    check ("gpr_$set_refresh_entry failed.", status);
}

/* process_x_events is the single input processing point for
   the application. */

process_x_events ()

{
XEvent  event;      /* event structure */
short   done = 0;   /* flag controlling main loop exit */
status_$t status;   /* gpr call return status */

    /* Continue to get and process events until the user
       presses the exit icon */

    while (!done) {
        XNextEvent( display, &event );
        switch (event.type)    {

             case ConfigureNotify:
             /* Update width and height globals when
                main_window is resized */
             if (event.xconfigure.window == main_window)
             {
                  /* update main_window size */
                  width = event.xconfigure.width;
                  height = event.xconfigure.height;
             }
             break;

             case UnmapNotify:
             /* Handle unmap gravity */
             if (event.xunmap.from_configure)
                  handle_unmap (event.xunmap.window);
             break;

             case Expose:
             /* Handle exposures */
             if (event.xexpose.count == 0)
                  refresh_window (event.xexpose.window);
             break;

             case EnterNotify:
             /* Hilite the icon when the cursor enters
                the icon window */
             if (hilite)
                  hilite_window(event.xcrossing.window,
                                colors[ICON_HILITE]);
             break;

             case LeaveNotify:
             /* Remove hilite when the cursor leaves the
                icon window */
             if (hilite)
                  hilite_window(event.xcrossing.window,
                                colors[ICON_BG]);
                break;

                case ButtonPress:
                /* Recolor the graphics window when the recolor
                   icon is pressed */
                if (event.xbutton.window == recolor_window) {
                   gpr_$acquire_display (&status);
                   check ("gpr_$acquire_display failed.", status);
                   do_gpr_graphics (True);
                   gpr_$release_display (&status);
                   check ("gpr_$release_display failed.", status);
                }
                else    /* Quit when the exit icon is hit */
                   done = (event.xbutton.window == exit_window);
                break;

                default:   /* Ignore other events */
                break;

            } /* switch */
        } /* while */
}


/* handle_unmap scales the unmapped window to fit the new
   dimensions of the main window and then maps the window */

handle_unmap (window)

Window  window;
{
int x, y, w, h;

     if (window == exit_window) {
         x = 10;
         y = 10;
         w = (width >> 2) - 20;
         h = (height >> 1) - 15;
     }
     else if (window == recolor_window) {
         x = 10;
         y = (height >> 1) + 5;
         w = (width >> 2) - 20;
         h = (height >> 1) - 15;
     }
     else if (window == apollo_window) {
         x = (width >> 2);
         y = 10;
         w = (width >> 2) * 3 - 10;
         h = height - 20;
     }
     else return;     /* don't fight with window manager */

     XMoveResizeWindow (display, window, x, y, w, h);
     XMapWindow (display, window);
}
```

```
/* hilite_window changes the background color of the window
   and redraws the graphics.
   NOTE: This is only called if hilite is supported. */


hilite_window (window, color)

Window          window;
unsigned long   color;

{
    XSetWindowBackground (display, window, color);
    XClearWindow (display, window);
    refresh_window (window);
}


/* refresh_window refreshes the window */


refresh_window (window)

Window  window;

{
    if (window == exit_window)
        draw_icon (window, EXIT_STRING,
                   EXIT_STRING_WIDTH, EXIT_STRING_HEIGHT);
    else
    if (window == recolor_window)
        draw_icon (window, RECOLOR_STRING,
                   RECOLOR_STRING_WIDTH,
                   RECOLOR_STRING_HEIGHT);
    else
    if (window == apollo_window)
        refresh_graphics ();
}


/* draw_icon draws a text string in an icon window. */


draw_icon (window, string, string_width, string_height)

Window  window;    /* icon window identifier */
char    *string;   /* text string to display in the icon */
int     string_width, string_height;    /* dimensions of the
                                            text string */

{
unsigned int icon_width, icon_height;
int xpos, ypos; /* to position string within icon window */

    /* Center the text if possible */

    icon_width = (width >> 2) - 20;
    icon_height = (height >> 1) - 15;

    xpos = (icon_width - string_width) >> 1;
    if (xpos < 0)
       xpos = 0;
    ypos = (icon_height + string_height) >> 1;
    if (ypos < ICON_FONT_CHAR_HEIGHT)
       ypos = ICON_FONT_CHAR_HEIGHT;

    /* Draw the text */

    XDrawString (display, window, icon_gc, xpos, ypos, string,
                 (int) (strlen (string)));
}


/* refresh_graphics:  after establishing a refresh entry
   routine, gpr will call that routine if necessary the next
   time the display is acquired.  Therefore, all that is
   needed to refresh the graphics is to acquire (and release)
   the display. */

refresh_graphics ()

{
status_$t   status;    /* gpr call return status */

    gpr_$acquire_display (&status);
    check ("gpr_$acquire_display failed.", status);
    gpr_$release_display (&status);
    check ("gpr_$release_display failed.", status);
}


void refresh_procedure (unobscured, pos_change)
boolean *unobscured, *pos_change;               /* ignored */
{
    do_gpr_graphics (False);
}

/* do_gpr_graphics draws seven concentric circles in each of
   the visible portions of the Apollo graphics window.  */

do_gpr_graphics (recolor_event)
Bool recolor_event;    /* true if this call is being made
                          because the user pressed the
                          recolor icon */
{
#define MAX_VIS_LIST_SIZE 64 /* no. visible clip rectangles */

gpr_$offset_t      size;
gpr_$rgb_plane_t   hi_plane;
short  vis_list_size;    /* no.of visible pieces of window */
gpr_$window_t vis_list [MAX_VIS_LIST_SIZE];/* visible pieces*/
short  clip_rect;           /* vis list loop counter */
gpr_$position_t  center;  /* center of the circles */
short  radius; /* radius of the largest circle */
```

```
static short first_color=0; /* first color index this time */
short   index;              /* index into colors array */
short   shrink;             /* radius shrink factor */
status_$t status;            /* gpr call return status */

    /* Determine the current size of the window */
    gpr_$inq_bitmap_dimensions(&bitmap,&size,&hi_plane,
                                                     &status);
    check ("gpr_$inq_bitmap_dimensions failed.", status);

    /* Get visible portions of the window */
    gpr_$inq_vis_list ((short) MAX_VIS_LIST_SIZE,
                    &vis_list_size, vis_list, &status);
    check ("gpr_$inq_vis_list failed.", status);

    /* If a recolor event is being processed, change the
       drawing colors */

    if (recolor_event)
       first_color = ++first_color % NUM_COLORS;

    /* Disable cursor echo while drawing */

    gpr_$set_cursor_active (false, &status);
    check ("gpr_$set_cursor_active failed.", status);

    /* Set up the parameters for the circle call */

    center.x_coord = size.x_size >> 1;
    center.y_coord = size.y_size >> 1;

    if (size.x_size < size.y_size)
        radius = size.x_size >> 1;
    else
        radius = size.y_size >> 1;

    /* Now draw into each of the window's visible pieces */

    for (clip_rect=0; clip_rect<vis_list_size; clip_rect++) {
            gpr_$set_clip_window(vis_list[clip_rect],&status);
            check ("gpr_$set_clip_window failed.", status);
            gpr_$clear ((gpr_$pixel_value_t)
                                 (colors[first_color]),&status);
            check ("gpr_$clear failed.", status);

            /* Draw some scaled concentric circles */
            for (index = first_color + 1, shrink = 1;
            index<first_color+NUM_COLORS;index++,shrink++) {

                gpr_$set_fill_value ((gpr_$pixel_value_t)
                        (colors [index]), &status);
                check ("gpr_$set_draw_value failed.", status);

                gpr_$circle_filled(center,
                                 (short)(radius/shrink),&status);
                check ("gpr_$circle failed.", status);
            } /* for */
        } /* for */

    /* Enable cursor echo again */

    gpr_$set_cursor_active (true, &status);
    check ("gpr_$set_cursor_active failed.", status);

}

/* terminate_gpr terminates GPR in the standard manner. */


terminate_gpr ()
{
status_$t status;

    gpr_$terminate (false, &status);
}


/* terminate_x terminates X in the standard manner. */

terminate_x (error_msg)

char    *error_msg;             /* optional error message */

{
    if (error_msg)
        fprintf (stderr, "%s\n", error_msg);
    XCloseDisplay (display);
}


/* check tests the status returned from GPR calls and
   terminates if it was bad. */

check (error_msg, status)

char    *error_msg;             /* optional error message */
status_$t   status;             /* gpr call return status */

{
    if (status.all != status_$ok )
    {
        if (error_msg)
            fprintf (stderr, "%s\n", error_msg);
        error_$print (status);
        terminate_gpr ();
        terminate_x (NULL);
    }
}
```

Glossary

Glossary

ADUS

The Apollo Domain Users' Society, which produces software shared among its members. ADUS offers kits of full source from MIT (contact the ADUS Administrator at Apollo).

background window

A shaded area (also called the root window) that appears behind or to the side of the other windows.

borrow mode

ADUS and MIT–tape Apollo servers operate in borrow mode, which means that the X server takes over the entire display.

client

An application program that connects to the window system server by an interprocess communication (IPC) path, such as a TCP connection. This program is referred to as a client of the window system server.

Color Table Manager (CTM)

Apollo's Color Table Manager allocates colors to pixels and stores this information in a "color map." Programs that do not use the CTM are not CTM–compliant and will probably interfere with CTM–compliant programs such as **Xapollo**.

cursor

A display symbol or pointer used to indicate the position in a window or screen corresponding to the focus of an input device. The cursor changes its shape depending on what kind of program you're running, your window manager, and where the cursor is on the screen. The shapes it takes can be any of the following.

- An "X" shape that appears in the root window.
- Right angle shapes that appear as upper-left and lower-right corners when you're placing a client.
- A circle shape (or target) that shows which window you've chosen for the **uwm** window manager to work with.
- An "I" shape that tells you where you are in an **xterm** window. (It's often called an "I-beam.")

display

A set of one or more screens driven by a single X server.

DM

Apollo's Display Manager that controls the display and input.

DM-owned root

The mode in which the DM acts as the primary window system, owning the background (root) window, drawing the cursor, performing window management functions, and so on.

event

Clients are informed of device input or client request side effects asynchronously via events, typically reported relative to a window.

font

An array of characters or other shapes, with additional information about their sizes and the spaces between them. The X protocol does no translation or interpretation of character sets.

glyphs

Images, usually of a character in a font, but also possibly some other shape such as a cursor, a tiny mailbox, or a pointing hand.

hints

Certain properties, such as the preferred size of a window, are referred to as hints, since the window manager will consider them, but it makes no guarantee that it will honor them.

icons

Icons are tiny windows that display mail boxes or other graphic symbols. They're used to hold "collapsed" windows so they don't take up much space. Even though you can't see activity in the window that has been collapsed, its processes are still active.

IDM

Apollo's Input Demultiplexor, a utility that controls workstation input from the keyboard and mouse, providing a mechanism to bind particular devices, events, and processes together.

input devices

The most common input devices are the keyboard and mouse that you use to indicate position in a window, enter text, select menu options, etc.

interludes

Subroutines that allow Pascal and FORTRAN programs to call Xlib functions.

keyboard *n*

Apollo's keyboards are numbered 2, 3, and 3a–3g. Keyboard 2, as opposed to the newer, low-profile keyboard 3, has no numeric keypad. (Keyboards 3a–3g are multinational keyboards.)

keycode

A code in the range [8, 255] inclusive that represents a physical or logical key on the keyboard. The mapping between keys and keycodes cannot be changed. A list of keysyms is associated with each keycode.

keysym

A symbol that is #defined in one of the keysym include files (**ap_keysym.h**, **keysym.h**, or **keysymdef.h**) and that is a portable representation of the symbol on the cap of a key. Each key may have several keysyms, corresponding to the pressed key when various modifier keys are also pressed.

mapped

A window is mapped if a call to **XMapWindow** or **XMapRaised** has been performed on it. This makes it eligible for display.

modifier keys

Shift, **Control**, **Meta**, **Super**, **Hyper**, **Alt**, **Compose**, **Caps Lock**, **Shift Lock**, and similar keys are called modifier keys.

native graphics

Apollo graphics routines and products such as 3D GMR or GPR that are not directly portable to non–Apollo hardware platforms.

raise

When you raise a window you change its stacking order so as to hide other windows beneath it.

reparenting

The process of assigning a different (often new) parent window to a child window. The window manager often reparents the top-level windows of each application in order to add a title bar and perhaps resize boxes (which are themselves windows).

RM

Apollo's Rectangle Manager that manages a workstation-global database of rectangular areas on the screen, determining their sizes, stacking order, ownership, etc.

root window

Each screen has a root window covering it. It cannot be reconfigured or unmapped, but otherwise it acts as a fully functional window. A root window has no parent.

screen

A server may provide several independent screens, which may or may not have physically independent monitors.

server

Provides the basic windowing mechanism. It handles connections from clients, puts displays on the screen(s), and sends input back to appropriate clients. It controls a single keyboard and pointer and one or more screens that make up a single display.

share mode

The **Xapollo** server allows the same screen to be shared between X clients and Apollo's native Display Manager.

window manager

A window manager arranges windows on a screen and controls the user interface for selecting which window receives input.

X-owned root

The mode in which X is the primary window system, owning the background (root) window and controlling the display and management of windows.

Index

Index

Files

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Using the X Window System on Apollo Workstations*
Order No. 015213-A02

Your Name ____________________ Date ________

Organization ____________________

Street Address ____________________

City ________ State ________ Zip ________

Telephone number (_____) ____________________

When you use X on Apollo systems, what **job(s)** do you perform?

- ☐ Programming
- ☐ Application End User
- ☐ Hardware Engineering
- ☐ System Administration
- ☐ Other (describe)____________________

How many years of **experience** do you have in using X:

____

What programming **languages** do you use with the Apollo system?

____

How would you **evaluate** this book?

	Excellent		Average		Poor
Completeness	1	2	3	4	5
Accuracy	1	2	3	4	5
Usability	1	2	3	4	5

Additional Comments: ____________________

____

____

No postage necessary if mailed in the U.S.

Cut or fold along dotted line

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

fold