Strategies for Porting the X v11 Sample Server

Susan Angebranndt

Raymond Drewry

Philip Karlton

Todd Newman

minor revisions by

Bob Scheifler

Revised for Release 4 and Release 5 by

Keith Packard

MIT X Consortium

X is a graphic user interface system enabling an application program to run on a computer other than the user's workstation. A "display server" (or simply "server") is the user's workstation. A "client" is an application program running on a machine with which the user interacts. Accompanying this document is the X Window System source tape, which also contains the "Definition of the Porting Layer" document.

This document is a guide to porting the X server software to your workstation.

1) Read the first section of this document (Overview of Porting Process). It describes what you will go through.

2) Skim the "Definition of the Porting Layer" document.

3) Scan the Details section of this document.

4) Start planning and working, referring to the Strategies and Definition documents.

   You may also want to look at the following documents:

- "The X Window System" for an overview of X.
- "Xlib - C Language X Interface" for a view of what the client programmer sees.
- "X Window System Protocol" for a terse description of the bytestream protocol between the client and server.
- "X11 Server Extensions Engineering Specification" for a description of how to add features to your X server in an agreeable way.

1. Overview of the Porting Process

1.1. Directories

This section describes some of the directories accompanying the tape.

lib/X
The X client library is used for developing client programs. Since it is simple and straightforward, it should be very easy to port.

doc/
This directory contains documentation, such as an online copy of this document and the others mentioned.

Makefile
This is a master make file for everything on the tape.

fonts/
This directory has font files in a text format and the compiler to compile them into a binary format that your server uses.

fonts/lib
This directory contains a library of routines for handling fonts. It contains all of the interfaces used by both the X and Font servers. It also contains routines for manipulating font files which are used by bdftopcf and mkfontdir.

fonts/bdf
This directory contains directories of font files. The font files are in a text format called "bdf." (See the "Definition of the Porting Layer" document, section 5.2.8.) There are three sections: misc, 75dpi and 100dpi. The misc directory contains fonts which do not fit a particular resolution; the cursor font, various fixed-pixel-size fonts and fonts which are needed by various toolkits for icon images. 75dpi and 100dpi contain identical font sets at 75 dots-per-inch and 100 dots-per-inch respectively. All of these fonts conform to the XLFD standard; most of them are in ISO-8859-1 (Latin-1) encoding.

fonts/tools/bdftopcf
This directory has the font compiler source. The old bdftosnf compiler had private definitions for the glyph format; bdftopcf inherits the definitions from the server so special configuration is not needed here. Besides, if you get it wrong, the server will still run, but will spend quite a bit of time reformating the fonts.

fonts/tools/mkfontdir
mkfontdir is run over a directory containing font files. It produces the database which the server uses to map font-names to file-names.

X11/
This is an important directory because it has some include files that are actually shared between client (Xlib) and server, such as X.h, Xproto.h. The files in this directory are actually installed from include/ during the initial build process.

server/
This directory contains all the source for the server code. There are some other files that are needed from other top-level directories, such as .h files from the X11/ directory.

server/ddx
This directory has all the Device Dependent X server code. Operating System dependent code is under server/os.

server/ddx/cfb
This directory has "Pixblit" and other code for Color Frame Buffer displays. (In this document, "color" means multiple bits per pixel. You may be able to simply plug in a few special numbers into defines, then compile it and have it work for your display. Otherwise, it will serve as no more than an example of what you should implement as Pixblit routines for your own frame buffer.

server/ddx/mfb
This directory has "Pixblit" and other code for Monochrome Frame Buffer displays. (In this document, "monochrome" means a black and white display with one bit per pixel. Even though the usual meaning of monochrome is more general, this special case is so common that we decided to reserve the word for this purpose.) If you have a monochrome screen, you may be able to simply plug in a few special numbers into defines, then compile it and have it work for your display. Otherwise, it will serve as no more than an example of what you should implement as Pixblit routines for your own frame buffer.

server/ddx/mi
This directory contains Machine Independent DDX code. It supplies the "Drawing Primitive" procedures that DIX calls to perform graphics and calls the "Pixblit routines." It also contains other routines. It is portable to any system that can draw graphics by manipulating rows of pixels by hand. See the "Definition" document for an explanation of these terms.

server/ddx/apollo,dec,hp,ibm,macII,mips,sun,tek
These directories contain device-specific code which runs on a wide variety of machines. All of these directories were, at least originally, contributed by the various hardware vendors. None of them represent actual production code entirely.

server/dix
This is the Device Independent source. You should not have to change any of these routines.

server/include
This directory has dozens of includes that are specific to the server. Many of them are pairs in the form of XXX.h and XXXstr.h, the former having #defines for the XXX topic, and the latter having struct definitions. This naming convention could not be carried out systematically as SysV allows only 14 character filenames, which truncates some of the XXXstr.h file names. The include files in this directory are only for DIX and DIX/DDX interfaces; individual modules which export functionality (such as mi) include the interface-definition header files in their own directories.

server/os
This directory has Operating System specific source, mostly in subdirectories.

server/os/4.2bsd
This is source for UNIX 4.2 BSD (Berkeley UNIX) source. It will also run on 4.3 BSD and ULTRIX. This code will also run on several vendors mixed SysV/4BSD systems. It provides many routines which are not very OS specific, but which haven't been moved elsewhere yet for lack of need (i.e. it runs on every device which is supported by the sample DDX directories).

This software is contributed to the public as a service. We welcome contributions from other development groups for inclusion on future distributions.

1.2. Areas of Work to be Done

Most of the code for the X server is on an industry standard 9 track magnetic tape in UNIX "tar" backup format. The missing parts that you must supply for your particular workstation fall into the following three categories:

operating system:
Your operating system is the first and most obvious source of differences. If you have a UNIX 4.2 or 4.3 BSD system, this part will be trivial to port. The further you move away from that, the harder it will be. For systems that are not UNIX-based, the hardest part of the porting process may be the interface to clients.

input:
You need to code specific interfaces for your particular pointing device (mouse or tablet) and keyboard. These have to be non-blocking; a scheduler must be supplied that can wait for input events and client requests to arrive.

output:
This is potentially the largest section of code you will need to write. If you have a memory-mapped frame buffer display, most of the code has already been written for you (but optimization may be desired). If you have color and/or a special-purpose graphics processor that insists upon doing all of the work itself, you have a substantial task.

1.3. About DDX, mfb, cfb, and mi

The DDX (device dependent X) layer provides a software interface to a conceptual hardware device. The imagined device provides primatives for drawing lines, arcs, text, filling areas, etc. These primatives may be actually provided in your hardware, or you may have to build them out of simpler primatives your hardware does provide. The mi (machine independent) routines provide software simulation of the conceptual machine built out of very simple primatives such as GetSpans, SetSpans, FillSpans, PushPixels, etc., which we call the Pixblit routines.

The mfb layer is one implementation of the software interface that connects to monochrome (one bit deep) framebuffers. The cfb layer is one implementation that connects to multi-bit framebuffers. In both cases, some functionality is provided by writing directly to the framebuffer. Some more esoteric functionality is achieved by calling the mi routines. In order to be able to use the mi routines, they must also implement Pixblit routines. Both cfb and mfb have been extensively tuned for Release 4 to run as quickly as portably possible on a wide variety of machines. Mfb, in particular, has some Gnu C Compiler "asm" statements which substantially increase performance of some operations on both Vax and 68020 CPUs. Cfb, on the other hand, can be tuned for new architectures by describing some CPU characteristics in server/include/servermd.h.

The mi code should be portable to all systems. It calls the Pixblit routines to apply the pixels, all device dependencies are contained in there.

Some routines in mi are not used by the mfb or cfb DDX implementations. They are provided to make it easier for you to get a simple port running quickly. Unfortunately, it is not possible to provide a complete DDX implementation in mi, you need the Pixblit routines which actually know how the hardware looks.

The mi, mfb, and cfb routines were designed for portability over performance. Therefore, you may want to spend time optimizing them if you choose to use them.

1.4. What do I do?

To start, you should get the simplest server running by modifying as little as possible, probably using mi and maybe using mfb or cfb. Later, you can carefully optimize it.

The first step is to copy the source code off the tar tape onto your machine. If yours is a UNIX system, this will be easy. If not, it may be more difficult.

Use the UNIX "tar" command to load the tape onto your machine, if appropriate. If you have a network running, you might be able to get it from some other machine on the net by using the UNIX "ftp" command (some non-UNIX systems also support ftp).

One way to load the source onto a non-UNIX system is to load it onto a UNIX system and move it to your system. If you are porting to a non-UNIX system, we strongly recommend that you have a UNIX system available in house for purposes such as this and for testing.

The next step is to create a subdirectory under the ddx and os directories as appropriate for your code. (See the "Definition of the Porting Layer" document for details on directories.) Copy files into these directories from sibling directories that seem closest to what you will need. For instance, if you are porting to an IBM 3279 display on an IBM 4361 mini, you create the directories ddx/3279 and os/4361 (or os/370 if you thought this would be portable to other 370 architecture machines). If you were porting to a 3279 display on a UNIX 4.2 system, you would make a directory ddx/3279 and use the os/4.2BSD directory the way it is, if you thought it would work. (If later in the process you found it did not, you would make your own subdirectory.)

Start modifying the code. Begin with the OS code. There are file i/o routines to work on, and the byte stream to the client is important. Get the byte stream working between your own test programs.

The second logical step is to get some form of the X server code running. Make dummy versions of the input routines and graphical output routines so you can concentrate on getting initialization right without having the system crash. Edit Xmd.h according to the instructions in the section "Machine Dependencies" later in this document. Then compile everything.

Next, work on the graphical output. Fill in whatever you need so that a simple client program that just draws some graphics on the screen works. For monochrome screens, setting a few defines and recompiling the mfb files may be all you need. (See "Porting MFB" in the Details, below.) For color screens, setting a few defines and recompiling the cfb files may be all you need. (See "Porting CFB" in the Details, below.)

The xclock program is a good candidate for testing graphical output. Depending on your networking software, it might be easiest to have this test client on the same machine as your server.

Finally, work on the input. Fill in code to handle the keyboard and mouse (or other pointing device). The cursor that echoes the position of the pointing device is better implemented in hardware, but mi does provides support for a primitive software cursor which is very easy to use. See the section on cursors below.

Next, optimize.

You are done! For more explanation, see the Details section, below.

1.5. Cost

We estimate that a basic monochrome or color server will take one to two months to develop if done on a UNIX 4.2 BSD system by an experienced C programmer who knows the hardware quite well.

The more software you have to write, the longer it will take. If it is a non-4.2 UNIX system, add one to four weeks. If it is a non-UNIX system, add one to two months. If your operating system does not have a network, that must be taken into consideration. If you buy someone else's implementation, add one to four months. If you decide to write it yourself, add six months to two years.

If special graphics hardware (a graphics processor, not just unusual bitplanes) is involved, it will take much longer. If you want the code optimized for maximum performance, it will take much, much longer.

The more experienced you are, the less time it will take. If you are new to C, add some time. If your programmer is not familiar with your operating system, it will take longer. If you are not familiar with windowing systems, it will take longer; if you're not even familiar with 2-d raster graphics, it will take longer still. If you've done ports to X before, it will take less time. If you are really hot, it will take less time.

Of course, all of these are just guesstimates.

The above figures are for one programmer. Some gains may be achieved through the parallelism of adding programmers. But, as Fred Brooks puts it, the bearing of a child takes nine months, no matter how many women are assigned.

If you do distribute the work, it would be best to devise a good partition. For instance, a reasonable partition might be to have one programmer work on the operating system, network and input code, have two more working on graphics output, with one of them concentrating on text graphics. We recommend no more than a few programmers at one time.

At any rate, if you have a product that is robust enough to be useful, you are probably about half way to making that product a solid, finished release.

2. Details

2.1. Tools

2.1.1. The C Compiler

Your C compiler can have a significant effect upon the time it takes you to finish the project. Since the original source was developed on a UNIX system, the closer your compiler approximates the UNIX C (pcc) compiler, the better. Depending upon your situation, it may be worthwhile to try more than one C compiler and use the one that works best. (Programmer time is quite expensive; software is frequently much less expensive, even if overpriced.) If, for instance, the DIX code does not compile without modifications, you may want to look elsewhere.

Sometimes we intentionally call a routine with the wrong number of arguments. For instance, there is a routine NoopDDA() in dixutils.c that is used widely as a procedure that does nothing. It has zero arguments but is used for situations where routines get passed different numbers of arguments. If this causes problems on your machine, you might need to change the code or get another compiler.

If you are using an 8086 architecture, we recommend you use "large" model to get the server running, then switch to mixed model for speed and space efficiency.

2.1.2. Make and Makefiles

"Make" is a UNIX program that manages the compilation process. It reads in a text file named Makefile describing the source files that need to be compiled and how. (This file is frequently called the dependencies file because it describes the chain of dependencies leading to the final product.) Make then checks the dates of source, intermediate, and object files, determines the minimum compiles needed to bring a given result file up to date, and runs each compilation step as a child process.

This idea has been imported to a wide variety of operating systems (frequently still called "make"). On non-multitasking operating systems, the program frequently generates just a batch file with the needed compile commands in it and then executes this batch file as its final operation. (Beware: few of these non-UNIX versions contain all the features of the original.)

We recommend using Make or whatever useful substitute you have available. The makefiles for the UNIX system are included with the tar tape, and they should work on any UNIX system. this code does not support "near" and "far" pointers. This may not be necessary or desirable on 386 systems. They might not work on your system. To aid you in generating your own makefiles for your own system, we briefly describe the syntax of makefiles.

The dependency relationships look like this:

```
fig.o : fig.c fig.h xyz.h
        cc -abc fig.c
```

This states that the file fig.o (an object) depends upon fig.c and the two .h files listed. If fig.o is found to be older than any of the dependencies, execute the command(s) listed below it to bring it up to date.

Most makefiles look much more complicated. This is primarily due to the use of macros. When you have a statement of the form:

```
COPTS = -abc -x fig -FPa
```

this means that you can subsequently use "$(COPTS)" as a text substitution macro elsewhere in the makefile.

```
fig.o : fig.c fig.h xyz.h
        cc $(COPTS) fig.c
```

This is frequently used as shown to hold C compiler options. It is also used to hold lists of filenames.

```
HFILES = fig.h xyz.h

fig.o : fig.c $(HFILES)
        cc $(COPTS) fig.c
```

Another common cause for confusion in makefiles is that there are special $ symbols that signify "the dependencies" or "the product" in a command line. These can be used in powerful constructs that will indicate, in just a few lines, "compile all .c files that you need to compile and do it this way."

Consult UNIX documentation for more details.

The makefiles supplied with the sample server are not guaranteed to be nearly as portable as the code. In particular, there are situations where special techniques were used to get everything to compile.

There are some routines that need to be compiled with #defines entered on the command line with the -D flag of the UNIX cc command instead of with a normal #define directive. If you don't have such a facility with your compiler, you should put such #defines in an .h file and do some file copying in the makefile to achieve the same result.

2.1.3. Debuggers

Because you are drawing graphics on the display, you will probably want to use a debugger that does not use the display. On some systems, a terminal connected to a serial port is the best way to communicate with the debugger. On network systems, you may be able to log into your test machine remotely and run the debugger and server from there.

2.1.4. Profiling Tools

After you have an initial implementation running, you may want to improve its performance. A profiler is invaluable for this purpose because it tells you where you are actually consuming CPU cycles. You can then change code based upon hard evidence. On UNIX systems, you might use the prof and gprof programs. To really analyze the code, it is very useful to use a basic-block profiler (like the MIPS pixie system); most of the frame-buffer graphics primitives are large functions wrapped around multiple small inner loops which perform the actual rendering.

To gain more insight into performance deficits, the X client "x11perf" (contrib/demos/x11perf), contributed by DEC, offers a wide array of measurement tools and an easy base to add more to. It was used extensively in the development of the current cfb and mfb drivers, along with the release 4 changes to various data structures and mi algorithms with frequently astounding revelations. It is hard to recommend this program too much. Each time you sit down to optimize some area of the server, first develop a test case and integrate it into x11perf; measuring before and after to discover performance changes. X11perf is also useful in profiling the server as it provides a repeatable sequence of graphics requests; set it up to use a fixed number of iterations.

2.2. Operating System Details

2.2.1. Machine Dependencies

The sample server is written to be portable to a wide variety of architectures, including CPU chips with different word sizes and different bit and byte ordering. Before compiling the code, you should set some defines to indicate what kind of CPU you have.

First, edit Xmd.h. Change the following:

INT32, INT16, INT8 should be signed integers of 32, 16 and 8 bytes. CARD32, CARD16 and CARD8 should be equivalent unsigned integers. BITS32, BITS16 and BYTE should be types that are most convenient for bit-oriented data. BOOL is the most convenient boolean value type that fits in 8 bits. Change them according to your compiler. Unfortunately, most of the mfb and cfb code "knows" that both int and long are 32 bits and will not work on systems where this is not the case. The rest of the server is less encumbered, but as the sample code has never been run on such machines, it is unknown whether it will work. It will certainly not work if long != 32 bits or short != 16 bits.

IMAGE_BUFSIZE is the size of a buffer of bytes that GetImage will return. Smaller systems may want to keep this at 1k or less; larger systems may put it at a few dozen k.

IMAGE_BYTE_ORDER indicates the order of bytes in the image. On VAXen, this is LSBFirst because the least significant byte is on the left, and is sent down the pipe first. On 68000s it is MSBFirst.

BITMAP_BIT_ORDER is the equivalent order of bits within a byte. On VAXen, this is LSBFirst because the least significant bit is most toward the left on the screen. On 68000s it is MSBFirst.

BITMAP_SCANLINE_UNIT is the biggest piece of memory in which IMAGE_BYTE_ORDER applies (in bits). For most hardware, 32 is a good value. Note that mfb and cfb both assume that addresses ascend across the screen from left to right and then proceed down the screen.

BITMAP_SCANLINE_PAD is the chunk size to which bitmaps sent over the bytestream should be padded. In other words, if you had a bitmap that only had one bit in it, would you want to send 8 bits, 16 bits or 32 bits?

LOG2_BITMAP_PAD must be the log base 2 of BITMAP_SCANLINE_PAD. If BITMAP_SCANLINE_PAD is 32, this must be 5.

LOG2_BYTES_PER_SCANLINE_PAD is the log base 2 of (BITMAP_SCANLINE_PAD divided by 8, the number of bits in a byte). If BITMAP_SCANLINE_PAD is 32, this must be 2.

2.2.2. Client Access

On many systems, one large section of code to be written may be the client access. X requires a reliable byte stream that can handle binary data. The sample server has code in it to communicate over three different byte streams: TCP/IP Ethernet, DECNET, and UNIX domain sockets.

If you do not have one of these already, you may find the byte stream somewhat time consuming to develop. If you have an operating system other than a UNIX 4.2 BSD system there is more work involved in client access. If it is another UNIX system, it is somewhat easier. The less it resembles 4.2 BSD, the more difficult it will be.

If you can't use TCP/IP Ethernet, DECNET or UNIX domain sockets, the alternative is to use some other byte stream mechanism. This will also have to be dealt with on the X client side (there is an implementation-specific routine in the X library to communicate with the server). You might start out by implementing both sides in the same machine as long as the client and server are separate processes and there is a convenient interprocess bytestream mechanism. In particular, this may be a first step toward implementation of your alternate inter-machine client-communication scheme.

In theory, any reliable byte stream will work. Its throughput should be approximately 5k bytes per second or more; otherwise performance will deteriorate.

For instance, an RS-232 or RS-422 link would work, although its performance would leave much to be desired unless you could achieve a baud rate of 56kbaud or greater. Since 8-bit binary data is regularly transmitted, your bytestream cannot use command characters for handshaking and protocol (such as XON/XOFF). Many modems or other telecommunication equipment will not work if designed for just normal ASCII communications because they may intercept certain control characters. Also, an RS-422 link would only offer one client-server bytestream, whereas you may want more than one such connection.

2.2.3. Multi-Processor OS's and Graphic Processors

The X server runs as a single process that imitates multitasking using an event-dispatching loop that checks for things to do from all sources and processes them one at a time. Many operations do not consume much time, so the multitasking appearance is upheld; but certain graphics operations may consume substantial amounts of CPU time. If another CPU or a graphics processor were available for these tasks, a significant gain in performance could be realized.

Graphics processors, in particular, can offer a unique opportunity to create a very high-performance X server. See the section "Implementing On Top of Another Graphics System" for more details if you have a graphics processor.

The X sample server was written as a single-threaded program for a single processor. A multi-processor system with a core processor (running the main server code) might dispatch tasks to a set of slave processors that effect low-level graphics operations. Or it may even have a completely different scheduling system, with multiple processors participating in the dispatch loop. In such cases, large parts of the server code will probably need to be rewritten. In particular, there are shared resources among clients, and you need to ensure that requests received by the server are executed in apparent synchrony, and you must ensure that global data structures such as the window tree and the resource table are maintained correctly.

X is merely a bytestream protocol and anyone can write any software to implement it in any language on any computer system. The sample server is merely one implementation.

2.2.4. Server Reset

The X server will reset itself immediately after all clients terminate. It is helpful to provide a way for the user to cause the server to terminate all client connections and reset itself. At an appropriate time, your server can cause all clients to be terminated by calling DoomClients(). The following cycle through the dispatch loop, all clients will be terminated in a somewhat reasonable way. This will cause a reset. Upon reset, you should instruct your network to close all open client connections.

For instance, when the server process receives a SIGHUP signal on UNIX systems, the signal routine calls DoomClients(). On a non-UNIX system, you may prefer a special sequence of modifiers and keys at the keyboard. Whatever the user does, all windows and applications will be closed and the user will have only an empty screen.

2.2.5. Shutdown

Depending upon your workstation environment, you may want your X11 server to run forever, or you may want to provide a way for the user to cause the server to quit gracefully without turning off the machine. Your server can quit by calling KillServerResources(), closing all network connections and then calling exit().

For instance, on UNIX systems, when the server process gets a SIGINT or SIGTERM signal, it calls KillServerResources() and then exit(). On a non-UNIX system, you may prefer to have the user press a special sequence of modifiers and keys at the keyboard. Whatever the user does to accomplish this, it will cause the X11 server to return to your operating system and/or shell. You may want to clear the graphics screen(s) before exiting.

2.3. Input Details

2.3.1. The WaitForSomething Scheduler

WaitForSomething() must wait for any of three occurrences: a hardware input event is received, a request from a client is received, or a request from a new client to open a connection is received. In the interim, you can do anything you want. On a multitasking system, you probably want to block yourself. This can be accomplished using mechanisms such as select(2) on 4.2BSD, or poll(2) on V.3. On systems on which the entire machine is dedicated to the X server you can loop endlessly, checking for input and client requests.

It would be unwise to depend exclusively upon idle times for polling the keyboard and pointing device. You should also poll these input devices at other times. In fact, these tasks should be monitored by an interrupt service routine checking at regular intervals. Otherwise, the users will be constantly annoyed when their keystrokes and mouse events are lost. Also, many paint-style programs depend upon regular pointing-device event-reporting to enable the user to draw smooth curves with the pointing device without leaps from one cursor location to another. (Even if the hardware can queue one or two such events, some graphic operations such as copying a large image can consume more time than a few keystrokes in rapid succession by a touch typist.)

DIX will process requests from each client until the variable isItTimeToYield is set. If you do not set it, you will enable one client to lock out all others by constantly drawing graphics. Therefore, you should devise a strategy for setting isItTimeToYield and ending the "timeslice" of a time-consuming client. The sample server will set this after ten requests have been read from the same client.

The DIX code will service each client in the order received from WaitForSomething(). You might tune the server so that if you write an event to a client, the priority of that client increases, by returning him earlier in the list or allowing more time before setting isItTimeToYield. You might set isItTimeToYield if the current request changes the window tree (causing exposures).

2.3.2. Keyboards

The keyboard consists of two kinds of keys, regular keys and modifier keys. Modifier keys, like Shift and Control, are keys the user presses while typing regular keys.

Your keyboard must be able to indicate when the user presses or releases keys. More specifically, your keyboard-interface software must be able to generate a KeyPress when a modifier or a regular key is pressed and a KeyRelease when a modifier key is released. You must also generate a KeyRelease for a normal key, but you can generate it immediately after the KeyPress is queued. If you cannot at least do this, you may have problems.

If your keyboard currently generates queue events upon each key motion or calls an interrupt routine that can do this, your situation is improved.

If you have a system in which a keymap has one bit for each key that is being pressed, you simply need to check this keymap at regular intervals in an interrupt service routine and queue events on an internal queue you maintain.

If you have a keyboard at the other end of a serial line, things become more difficult because you must reverse-map your ASCII characters into keycodes. In addition, you need to simulate modifier keys being used. For instance, when you get a lowercase "a", you must send a KeyPress for the "A" key, then a KeyRelease for "A". If you get an uppercase "A", you must send a KeyPress for the Shift key, send a KeyPress for the "A" key, then a KeyRelease for "A", then a KeyRelease for Shift. If you get a space character, you do not know if the shift key has been pressed, so you assume it has not. Between keystrokes, there is no way to know if the shift key has been pressed. Since with this scheme the client cannot ascertain when the user is pressing the shift key without typing any keys, some client applications that try to detect this will not operate properly.

If you want autorepeat, you must simulate this in your code or hardware by generating KeyPress and KeyRelease events when appropriate. The X protocol specification describes in detail how these options are set by a client.

2.3.3. Pointing Devices

The pointing device may be a mouse, a graphics tablet, a light pen, a touch screen, a trackball, a joystick, a pair of thumbwheels, or any other device that allows the user to indicate a location on a two-dimensional surface. The surface should bear some resemblance to the screen, because a visible cursor is displayed on the screen at a location that corresponds to the pointing-device location. The pointing device must report a location as a graphics coordinate on the screen.

The pointing device must have one or more "buttons" or other momentary control that the user can touch or press, such that the software driver can report a "press" and a "release" event. For instance, a touch screen can report press and release events when the user touches the screen. A trackball will probably require one or more separate buttons.

Some of these pointing devices are absolute, some are relative. For instance, with a touch screen, the user directly indicates the desired location on the screen. Mice and trackballs, on the other hand, only provide relative motion information; some other hardware or software must integrate these moves into a location. A graphics tablet is on the absolute side, but requires a mapping between the absolute coordinates on the tablet surface and the screen coordinates.

Some relative devices, such as mice, have a scheme in software or firmware to "accelerate" the motion of the mouse. For instance, on the Apple Macintosh, the interrupt service routine for mouse motions checks each increment to be added to the cursor location. If the jump is past a certain threshold, it doubles the jump distance. In this way, the user can move the mouse quickly across the screen, while still retaining fine control over the location for detail work. Unfortunately, this technique is frequently used because the hardware simply cannot generate fine enough position increments. If you implement or have available such a scheme, you should allow standard control calls from a client to turn this effect off and on.

Buttons are numbered starting with one. Probably, the left button on a mouse should be number one and they should be numbered towards the right from there. Client applications that use fewer buttons than you have will start with one and use only as many as needed. Since the X protocol specifies mechanism, and not policy, programs that depend upon more mouse buttons than you have may end up waiting for a long time before you hand it a button click which you cannot generate. On the other hand, light pens, graphics tablets with pens, and touch screens all implicitly have one "button", so it is reasonable to assume that client developers will be encouraged to consider one-button pointing devices.

Keep in mind that the mainstream pointing devices will be mice with one or more buttons and graphics tablets. Client programs written with one pointing device in mind may prove hard to use with another pointing device. That is, programs written for a mouse usually assume that the mouse location can be chosen very accurately. If your touch screen is coarse, it may be very frustrating to use. Also, a touch screen usually cannot generate mouse move events while the mouse "button" is not "pressed".

Make a mouse in a multiple screen environment move from one screen to the next by creating the impression that the screens are adjacent to one another; when the user moves the pointing device off the edge of one screen, the cursor moves onto another. X provides no policy for this, and you are free to make any geometric models you please.

2.4. Graphics Output Details

2.4.1. Porting MFB

If your screen is a simple monochrome frame buffer, you probably want to start by porting the mi and mfb routines. These will get you up quickly so you have something that works on which to build. Mfb has been extensively tuned for a few environments; in particular mfb runs very well on 68020 and vax CPUs where GNU CC is available. It also runs quite well on many RISC processors, where C compiler technology is more able to optimize some of the common operations. Although you could easily expend considerable time optimizing it, it is not unreasonable to leave most mfb routines the way they are.

The mfb routines are extremely portable. Most monochrome screens need only a half-dozen defines changed before the code works. System bit and byte order and other machine dependencies are given by #defines. (It assumes that byte ordering on the screen is the same as byte ordering in main memory.)

First, make sure you have edited Xmd.h for your CPU. See the section "Machine Dependencies" for instructions on how to do this. Then edit server/include/servermd.h to set up bit/byte orders and font padding information (mfb will work with any font padding, but MSBFirst machines work best with GLYPHPADBYTES == 4, GETLEFTBITS_ALIGNMENT == 1).

Next, write a screen initialization routine which sets the whitePixel and blackPixel values in the screen structure and calls mfbScreenInit with appropriate parameters; in particular you'll need to pass the address of the frame buffer, the screen size in pixels, both horizontal and vertical resolution in dots/inch (truncated to an int, which limits the accuracy a bit) and the frame buffer width in pixels. This last parameter may seem redundant, but many displays have extra framebuffer memory per scanline which is not visible on the display. Set this final parameter to the total pixel width of the display and mfb will ignore the invisible space. You could just type in a literal address in hexadecimal for the frame buffer address, but you may want to be a bit more sophisticated.

In this screen initialization routine, you'll want to initialize the various screen functions which apply to your hardware; hardware cursor routines (or mi software cursors) should be set up here. Mfb requires that the CloseScreen function which it stores in the screen be called at server reset time, make sure you wrap it if you need your own hooks here. If mfbScreenInit returns without troubles (TRUE), call mfbCreateDefColormap(pScreen) to initialize the default colormap with appropriate values.

That's it! All other machine dependencies should be taken care of, for most screens.

If you have an interlaced screen, where rows of neighboring pixels are not neighboring in memory, there is a way to make mfb work on it. The changes needed are few; carry them out carefully. They involve changing the mapping from the row number to address. Look for places where we multiply by devKind or width.

2.4.2. Porting CFB

If your screen is a simple packed-pixel frame buffer (either gray scale or color), you will want to start by porting the mi, cfb and mfb routines. You'll need to use mfb, even though your screen is color, as each server is required to support 1-bit pixmaps. The cfb routines have been extensively tuned for 1-byte-per-pixel displays and will work quite well with little change. On other displays (2 bit up to 32 bit), the existing code will still work, but in many areas performance will be disappointing.

The cfb routines are also extremely portable. Most color screens need only a few changes to Xmd.h and server/include/servermd.h. The 8-bit specific cfb text code works best with GLYPHPADBYTES == 4, GETLEFT_BITS_ALIGNMENT == 1, but will function with any padding. Also in servermd.h are several CPU specific tuning parameters. Read the comments carefully at the top of the file and set the ones appropriate for your CPU. They do not affect the correctness of the code, but can offer substantial performance gains if set correctly. If you are unsure, guess and use a profiling tool to discover which set work best. As with the mfb code, you could spend almost unbounded effort tuning various portions of the cfb layer for your particular system; but most of the code should run well enough unchanged to not warrant the effort.

Finally, set up an initialization routine which calls cfbScreenInit which uses arguments similar to those used by mfbScreenInit, the sizes are all still in pixels. CfbScreenInit does take an additional parameter before the framebuffer width, the visual class of the default visual. Set this appropriately (probably PseudoColor). You needn't set up whitePixel/blackPixel on pseudo color machines as cfbCreateDefColormap() will pick appropriate values and store them in the colormap when called after cfbScreenInit returns success. If you want to force the values for whitePixel/blackPixel, set them in the screen structure after cfbScreenInit and before cfbCreateDefColormap.

2.4.3. Implementing On Top of Another Graphics System

Many workstations already have their own graphics library or even their own windowing system. In order to coexist with the rest of the world as peacefully as possible, you may want to implement your X server on top of such a library. In fact, your machine may come with its own graphics processor that can greatly speed up graphics if used judiciously. Beware, however, that many X clients draw small objects, or only a few at a time. The overhead for translating X requests into graphics-system primitives may dominate the drawing time and cause the resultant server to be slower than a simple dumb frame-buffer system. Do not casually assume that the graphics processor is the fastest way to do things. Profile, profile, profile.

Since such graphic systems usually perform high level operations such as line drawing, text drawing, and area fill, you would start accommodating them at the "Drawing Primitives" level. In other words, you would rewrite one or more of the drawing primitive routines provided (such as miPutImage(), miPolyArc(), miPolyFillRectangle(), or miImageText8()). Instead of using the equivalent mi routine, you would write your own routine to use the graphics system.

One problem with a graphics processor, which also occurs when trying to implement a server atop an outside graphics library, is that the definition of certain functions can change in subtle ways.

For instance, a graphics processor may support text drawing only by ORing the glyphs into place; the X routines require more sophisticated text-drawing capabilities. A more difficult case is that in which a graphics processor can draw only fixed-width characters or can draw only 8-pixel-wide characters, or can draw characters only in its own hardwired font.

There are several approaches to this problem. First, you can recognize the 80 percent of the situations that can be executed by your graphics system, using the graphics system for those cases, and then executing the remaining 20 percent with mi (and possibly even cfb or mfb) code. Your GC validate routine can route different requests to various routines to do things differently. (See the Definition document for more information on the GC validate routine.)

Secondly, you can supplement the graphics processor's work. You can implement each X primative call for with more than one call to your graphics system, possibly with some auxiliary touch-up.

Third, request changes in your graphics processor or library.

By using as many of these approaches as appropriate, you can maximize the overall performance and compatibility of your workstation while correctly interpreting the X protocol.

Example: Your graphics processor applies glyphs only by "ORing" them into the image. Make the ImageGlyph routine call the graphics processor to draw the character's rectangle in the background color, then call the graphics processor to draw the character. If using just a solid-fill style in OR mode, you make the PolyGlyph routine call the graphics processor to draw the character. You use the slower mi routines for PolyGlyph routine that must effect tiling, stippling, etc.

Example: A graphics processor can draw only fixed-width characters. In this case, you use the Validate routine to change the primitive procedure pointers in the GC depending upon whether your font is fixed width or variable width. The fixed-width fonts go directly to the graphics processor. The variable-width fonts would be drawn in software, probably using routines borrowed from the sample server. (Depending upon the application, much text on the screen may be fixed width in the default font.)

Example: The graphics processor cannot clip to an irregular region as the entire Drawing Primitive set must do. Each routine checks the clipping region and ascertains whether the entity to be drawn falls entirely within the region. If so, the drawing is executed by the graphics processor. If any part of the entity is clipped, it is handled by the mi, cfb or mfb code.

Example: A graphics processor can draw text only with its own hardwired font. You create the font data that would correspond to your hardwired font, including the character glyph images. You make up a name for this font and make that your default font. Once again, you use the Validate routine to change the primitive procedure pointers in the GC depending upon whether your font is the hardwired font or not. The hardwired font goes directly to the graphics processor, as long as you can handle the fill style and clipping. Other fill styles or clipping may be handled by using hardware to draw into a pixmap and then applying it to the screen. Anything else would be drawn in software, probably using routines borrowed from the sample server.

Example: In X, lines are drawn with a model borrowed from PostScript in which the width of a line is a scalar number and ends of lines can either be butt (squarely cut off perpendicular to line) round (semicircular end), or projecting (like butt but extending past end of line by 1/2 line width). Imagine your graphics processor draws lines by smearing a rectangle from the source to destination. You get to set the height and width of the rectangle, but nothing else. You will not be able to use this operation for X wide lines in any but the simplest (i.e. horizontal/vertical or zero-width) cases.

In X there are few requirements placed on zero-width lines. (If you get a line width of zero, the intent is that it be "the fastest, easiest line," not an invisible line that has no width.) Fill-style rules still apply, the width should be approximately 1 pixel. The line style (dash style) should still be processed. The join style can be ignored because all join styles look the same at this resolution (except that miter joins for acute angles can get very long; you can ignore this effect). Your algorithm can be anything reasonable, it is desirable that you obey the cap style "NotLast" which indicates whether the ending pixel should be drawn. There is also a requirement that the lines be identical in the face of clipping; and a suggestion that the lines be identical when drawn in the reverse direction. Client programs that are picky about the lines they draw can draw width 1 lines. Your GC Validate routine can change the line-drawing routine pointer in the GC so that zero width lines get drawn by the graphics processor and the others are drawn by mi.

Of course, the facilities of each graphics processor are unique and each has special considerations. This is an area that will require meticulous attention to detail on your part.

2.4.4. Hardware Tiling and Stipples

Some hardware has the ability to apply patterns to the graphic surface. X makes a distinction between a tile versus a stipple. A tile is a "full color" pattern, the depth of which matches the target drawable. A stipple is a binary pattern that writes the foreground color where there are 1-bits areas and (if opaque) the background color on 0-bit areas. In addition, X allows a tile or stipple cell to have any size.

Some graphics processors can apply patterns that are only certain cell sizes, such as 8x8 or 16x16. Most CPU chips will apply patterns more efficiently to some frame buffers when the pattern width is 8, 16 or 32. In these cases, you use the GC validate routine to switch between fast pattern writing versus slow pattern writing via the mi routines. If your pattern size is a factor of your hardware pattern size (such as 2x4), you can simply replicate it to fill the hardware rectangle. (Many patterns will, in fact, be such sizes, so this will not be wasted effort. There is a request, QueryBestSize, that a client can execute to ascertain what sizes are optimal.)

2.4.5. Graphic Contexts in Hardware

Many hardware and firmware graphics systems have internal state analogous to X's Graphic Contexts. Such settings as current line width, current font, and current foreground color can be set in hardware for subsequent drawing operations. The sample server provides a mechanism for conveniently and efficiently specifying these settings: the GC validate procedure, which is called when necessary just before drawing.

Each drawable (window or pixmap) has a fixed serial number, which is unique for that drawable. Each GC has a serial number field that reflects the last drawable for which it was validated. Before a drawing operation with a drawable and a GC, the two serial numbers are compared; and, if different, the validate routine(s) are called to validate the GC.

When a GC is validated for a drawable, its serial number is set to the serial number of the drawable so that the next time these two are used together, the validate routines are not called. But the GC serial number is changed when some of its fields are changed, forcing a validate the next time around (the high bit is changed- it is unused for anything else).

In other words, by default this validate procedure you write is called only when the graphic context about to be used in a drawing operation has been changed since the last validate for this GC and drawable or if the last validate for this GC was for another drawable.

If you have only one hardware GC state, however, the validate routine must be called more often, because it must also be called whenever you switch between different GC's. For instance, under normal conditions, if you drew with drawable a and GC A and then drew with drawable b and GC B and kept switching between aA and bB without changing the GC's, each would no longer need to be validated because their serial numbers would match.

You could ensure that the validate routines are called for each change of the GC in use by keeping a static GC pointer variable that points to the last GC used. When a new GC is validated, the serial number of the last GC would be changed (change the high bit -- do not change the rest which is clipping information). Once this has been done, set your static GC pointer to point to the new GC. The validate routine will then be called whenever the hardware GC information needs to be changed. Unfortunately, the validate routine is probably much more involved than is necessary for this process. Instead, keep a global variable which points at the current GC in use and check in each graphics operation that the global GC pointer matches the GC passed in. If not, call a function to reload the hardware state from the new GC and change the global pointer to point at the new GC. DestroyGC would then check to ensure the cached GC pointer was invalidated when the GC was deleted.

If you have a sophisticated graphics processor that has, for instance, eight "contexts" of graphic parameters among which it can switch, you can retain eight static GC pointers (in an array). Before each graphic operation, set the hardware to use the hardware GC it needs. (You might want to run benchmarks to ensure you are not spending more time switching hardware GC's than necessary.)

See the Definition document for more details.

2.4.6. Implementing X on top of Another Window System

If you have another windowing system on top of which you want X to run there are several procedures in the ScreenRec and WindowRec you can use to execute almost all window operations. (Remember, DIX does not interact with your screen directly, so there is considerable leeway in this area.)

For instance, the window borders are always drawn with PaintWindowBorder() and the background with PaintWindowBackground(), which you supply. The contents of windows are drawn with the Drawing Primitives, which you supply. In addition, DIX calls your routines CreateWindow() and DestroyWindow() when it makes and destroys windows. Other hooks are provided for mapping and unmapping windows, moving them, and changing their attributes.

See the Definition document section on windows for more details.

2.4.7. Color

Color requires special considerations. You need to decide what class of display you have (see the Definition document, the section on Visuals and Depths).

Next, set up all of the visuals you will support. Each depth can have one or more visuals with which it is associated; if your screen has several modes, you can list them all. As with depths, it may be best to begin with the simplest and then add visuals one at a time.

Cfb has quite a range of support for colormaps. It has routines which emulate any visual type on a pseudo color system, most of which are also appropriate for other hardware types. You'll need to implement StoreColors, InstallColormap, UninstallColormap and ListInstalledColormaps; all of which are typically quite short. Place pointers to these routines in the screen structure before calling cfbScreenInit. If you are not using cfb, you may want to extract the colormap code anyway; it is not dependent on the rest of that directory.

You might want to construct your server so that it appears to support multiple lookup tables simultaneously, so you can have multiple Colormaps installed at the same time. For instance, if you had a display that had ten bits per pixel and a lookup table of 1024 entries, instead of declaring the obvious, you could declare that you had a display with depth 8 and four lookup tables. The extra two bits in each pixel would determine the lookup table to use for that pixel. Each time you wrote into windows on this screen, you would need to write those extra two bits surreptitiously to indicate the lookup table to use for this pixel. When copying pixel data off the screen onto pixmaps, the window would be considered eight deep, the extra two bits would be ignored. CopyWindow() would have to attend to these extra bits as it changed the colormap allegiance of affected pixels.

2.4.8. Multiple Screens

If you have multiple screens, the implementation is more complicated. Each screen may have its own method of managing windows or drawing graphics.

Each screen may have a different scheme for its frame buffer. Each screen manages pixmaps whose format is specific to that screen. There are no commands available to the client to transfer pixels directly from one screen to another or between pixmaps of different screens.

Each server must decide what depths and formats of image pixmaps it is willing to transfer between the client and server. This usually involves some consensus among the screens. A given server must support depth 1, and probably supports all of the depths of its screens.

Fortunately, you need not implement routines to copy pixels between different depths. The only way for the client to copy pixels between drawables of different depths is with CopyPlane, which copies one plane from one drawable to another. The client can copy whatever planes it needs into 1-deep pixmaps and can then logically combine these to achieve any desired result.

Every drawable has a fixed depth. Every GC has a fixed depth. The GC's depth must match the depth of the drawable for drawing, or an error results. Any tile pixmap used with a GC must be the same depth as the GC.

All screens should have the same byte and bit ordering. If they don't, you need to declare the "real" bit and byte ordering to follow one of your screens and set the variables in the screenInfo struct to it. Conversion would happen in GetImage() and PutImage() for each screen.

2.4.9. Backing Store and Save-Unders

Backing Store and Save-Unders are schemes in which the server saves parts of windows concealed by other windows so that when they become exposed again, the server can replace the pixel values quickly instead of asking the client to repaint the window.

Backing Store is a scheme where a window stores away obscured areas of itself when covered by other windows. Save-Unders is a scheme where a window saves away parts of the windows beneath it when it is placed in front. The basic idea is the same, but the subtle differences have important implications.

With Backing Store, a window tracks its own contents. When the client draws into a window that is partially obscured, the window must intercept these drawing operations and either cause the drawing to happen to the saved backing or forget the saved backing so that an expose event is generated the next time that part is exposed.

With Save-Unders, this is difficult because the window would need to know which pixels are associated with which windows; it would need to intercept all drawing commands to all windows. For this reason, Save-Unders is practical only for situations in which either there will be no drawing underneath, or if there is, it can be easily intercepted in one location in the code. (See the section on software cursors for an example of this.)

Backing store, on the other hand, is more complicated in another way-- the pieces of backing that need to be stored are often irregular shapes. In the case of X, windows are always rectangular, so the backing store can always be saved as a set of rectangular pixmaps. If this is done, though, drawing into the backing becomes extremely complicated and probably slows the system to the extent that your initial performance savings are severely diminished. If backing is saved as one large pixmap, you waste pixmap memory; you essentially retain a duplicate copy of each window in memory in which the only parts that are not used are those exposed on the screen.

Thus, it is usually most practical simply to discard parts of backing store that are drawn onto while hidden; an expose event will always execute properly.

The sample implementation of backing store is very device-independent. All that is needed to use it is a small vector of device-specific functions, only two of which are typically used; SaveAreas and RestoreAreas:

```
(*miBSFuncs->SaveAreas) (pixmap, region, x, y);
        PixmapPtr pixmap;
        RegionPtr region;
        int x, y;

(*miBSFuncs->RestoreAreas) (pixmap, region, x, y);
        PixmapPtr pixmap;
        RegionPtr region;
        int x, y;
```

(*SaveAreas) copies the specified region (which is pixmap relative) from the screen starting at (x,y) to the pixmap; (*RestoreAreas) copies the specific region (which is screen relative) from the pixmap to the screen, starting at (x,y). If you can provide these two functions; call miInitializeBackingStore(pScreen, funcs) and the rest will be taken care of. Cfb and mfb already call miInitializeBackingStore.

DIX provides SaveUnders when DDX provides BackingStore. This implementation is not as optimal as a real save unders implementation would be, but is better than nothing in most cases; the mi backing store implementation changes its behavior when bits are saved because of save unders instead of backing store.

2.4.10. Software Cursors

The sample server is designed for a hardware cursor that maintains a separate cursor bit map in hardware so that the video electronics mixes the image of the normal display and the cursor before being displayed. Nevertheless, a software cursor module is provided with hooks which require various levels of support.

The easiest to use level is the DispCur module (server/ddx/mi/midispcur.c). This provides software cursors with nearly no device-specific code. All that is required is that you read events from the pointer device and send position update events to the mi routines. Four routines are required, one of which is implemented in mi for the truly meek. The relevant header file is "server/ddx/mi/mipointer.h"; this file contains the function vector definition and some useful function defines.

```
typedef struct {
    long     (*EventTime)();         /* pScreen */
    Bool     (*CursorOffScreen)();   /* ppScreen, px, py */
    void     (*CrossScreen)();  /* pScreen, entering */
    void     (*QueueEvent)();  /* pxE, pPointer, pScreen */
} miPointerCursorFuncRec, *miPointerCursorFuncPtr;
```

An initialized structure of this type is passed, along with the screen which needs cursors to miDCInitialize(pScreen, &pointerCursorFuncs). (*EventTime) is required to return the time of the last event processed (as 32 bits of milliseconds). This allows the mi cursor support to build events and fill in the appropriate data.

(*CursorOffScreen) is called whenever the cursor would be off of the current screen if the user motion were tracked exactly. This routine returns FALSE if the cursor should be confined to the screen, TRUE if cursor should wander to some other screen. ppScreen should be smashed to indicate the new screen, px and py should indicate the position on that screen; they are initialized to be the position of the cursor on the old screen if the cursor were not confined or warped (i.e. (x,y) is not on the screen).

(*CrossScreen) is called whenever the cursor is moved on/off of the screen. entering is TRUE when pScreen is the screen now containing the cursor and FALSE when pScreen used to contain the cursor.

(*QueueEvent) is called in response to WarpPointer protocol requests. It should place the event at the tail of the input queue to be processed in series with the other events; this is frequently quite difficult to implement, however, and the mi routine, miPointerQueueEvent, simply processes the motion event immediately can be used (this may cause occasional small protocol violations).

When your pointer device moves, call

```
miPointerDeltaCursor (pScreen, dx, dy, generateEvent)
        ScreenPtr pScreen;
        int dx, dy;
        Bool generateEvent;
```

with the distance the device has moved and TRUE for generateEvent. If you device reports absolute coordinates instead, use miPointerMoveCursor instead (which replaces the delta coordinates with absolute ones).

To fill in the current pointer position for other event types, use miPointerPosition (pScreen, &rootX, &rootY), passing the address of the event rootX, rootY fields which will be filled in as appropriate.

The other two cursor layers can be investigated by looking through the miDispCur layer which uses them. In particular, you may want to use miSprite which allows you to provide device-specific cursor drawing primitives to speed up cursor rendering.

2.4.11. Limited Hardware Cursors

Many hardware cursor systems limit the maximum size of the cursor (for instance, to 16 pixels square). The X specification, however, specifies that a cursor can be any size. It is allowable for the server simply to truncate the cursor to an appropriate n-by-m rectangle. This may be the top-left corner, or it may be any n by m pixel rectangle that is entirely within the cursor and contains the hotspot; the exact choice is implementation dependent.

2.4.12. Fonts in Off-Screen Memory

Fonts are probably stored on disk on the server when not in use, probably in a bitmap format in binary, a form that is ready to go. Character drawing consumes much of the CPU, so you should try to ease the burden.

Of course, you need to read fonts into memory when they are needed. Unless you have an extra megabyte of main memory, it is probably best not to retain them in memory forever; users have a tendency to build up large font libraries.

You should have some scheme for loading fonts into memory on demand and for purging old fonts when no longer needed. Rarely will people use more than a dozen fonts simultaneously. (The main exceptions are programs specifically designed to show a sample of each font and novice What You See Is What You Get word processor users.) You will probably want to record the font least recently used and purge it when required. Appropriate algorithms can be found in many places, or you can devise your own.

The binary format in which the fonts are stored (probably snf) has glyphs aligned and padded to byte, 16-bit, or 32-bit boundaries. You can decide which based upon #defines.

2.4.13. Graphic Memory Usage

Some servers have extremely complex hardware, possibly consisting of multiple frame buffers among which the screen can switch, possibly having a graphics processor. Sometimes, the graphics processor has its own address space that may include memory in addition to the frame buffer that is displayed on the screen. Sometimes, the graphics processor can also access main memory in your server. Sometimes, your main processor can access graphics-processor memory. Sometimes, your main processor cannot access the frame buffer.

For these situations, you should carefully consider what to put in graphics memory and what to put in main memory for your particular hardware configuration. You should consider putting the following in graphics memory:

- Anything you must put in graphics memory because of the requirements of your graphics processor
- Hardware color lookup tables
- Hardware GC information
- Cursors
- Font Glyphs
- Pixmaps
- Regions
- Save-Unders
- Backing Store

Use the GC validate routine to move things in and out of graphics memory.

If your graphics hardware has limited resources, you might want to consider drawing into pixmaps that live in main memory, rather than special graphics memory. To do this, you should provide an in-memory version of the Spans functions. When drawing to an in-memory pixmap, and swap these Spans functions and the mi output code into the GC at ValidateGC time. Then the mi code will draw the appropriate things into the bits in memory. This will probably be slower than using the graphics hardware, but may be easier that dealing with memory allocation on the graphics hardware.

Furthermore, you might consider drawing into pixmaps in main memory if your hardware does not draw according to the X11 spec; mixing the two styles of drawing may produce odd results.

After you have implemented the above, and you use your X server, reconsider your decisions. (It is difficult to know how you will use an X server before you actually do so.) You may find that you want to change the use of graphics memory.

2.4.14. Graphic Output Tuning

The mi code is designed to be portable by sacrificing a certain amount of performance. Once you have got it running and have a large user base, it might be appropriate to make it run faster. Mfb and cfb have already been extensively tuned for many platforms; it is unlikely that you could increase performance by substantial amounts without resorting to assembly or gratuitous code expansion.

The overall rule in optimizing software is to collect experimental data. Do not subjectively judge whether something "feels" faster; subjectivity can be easily led astray. Do not merely assume where the performance bottlenecks are: use a profiler; run benchmarks; use a stopwatch.

If you do not have a profiler, try running a series of benchmarks. For instance, if you think that a major bottleneck is a certain loop in ImageGlyph, try commenting out the loop to see what performance gains are effected. Run benchmarks before and after, while running a program that will exercise that function. This gives you an indication of whether your hunches are right concerning the location of the bottlenecks before you devote a great deal of time implementing and debugging a complex algorithm.

Before you install an optimization, run benchmarks. After you install the optimization, run the benchmarks again to check performance gains. Complicated software that yields no substantial performance gains will simply be a liability later when the software needs to be modified.

Much optimization effort should be directed toward the operations that are executed most frequently. Sometimes, you can make a quick routine to handle a special case that occurs frequently and leave the more unusual cases for more general software that takes the time to handle all cases. For instance, most items that are drawn will be entirely within the clip region. Most of those that are not will be entirely outside of the clip region. Most drawing is executed with a plane mask of all 1s, and with an alu mode of Copy. Most drawing is done with a solid fill style. If draw is done with another fill style, the tile or stipple frequently has a size that is a byte or word multiple. The cfb and mfb routines have already been optimized for some of these special cases.

In general, start optimizing where you have a better algorithm or know more about the hardware than the portable routines.

2.4.14.1. First-Round Optimization

The most important things to optimize first are probably text drawing, zero-width lines, and large area pixel copying and filling.

Text drawing is best optimized by working on the Glyph routines. You may want to rewrite them in assembly language or implement them in hardware. Since most glyphs are written with solid fill styles and the glyph images usually do not lie on a clip-region boundary, you may want to make your speedy routine handle just this special case, and handle everything else with mi, cfb and mfb routines.

You can even optimize the mfb and cfb glyph routines to your machine without changing much. Fonts glyphs are padded to byte boundaries for each scanline. You can have this padded to 32-bit boundaries, if desired. The macro getleftbits() in maskbits.h gets glyph bits from glyphs; optimize it for your machine. (For instance, take into account byte, word and longword boundaries, whether your machine can address 16-bit or 32-bit words, and whether this is efficient.)

Zero-width lines are a good candidate because the rules for drawing them are relaxed. You need not worry about many of the details. Frequently, hardware or firmware can generate these. The most common lines are vertical and horizontal; special routines to draw these may be worthwhile.

CopyArea and CopyWindow optimization will improve window-movement performance. Frequently, a machine will have special hardware to perform such graphic operations.

2.4.14.2. Second Round Optimization

The next phase of optimization will probably concentrate on painting window backgrounds, wide lines, some of the easy-to-perform rectangle operations, and PushPixels().

Wide lines no longer present much of an opportunity to invest a great deal of work into an optimization and receive much benefit from it. The mi code now uses integer arithmetic for nearly all of the wide line computations and provides nearly-perfect protocol-conforming lines. Experimentation with moving the rendering into cfb or mfb has shown that not much performance is to be gained; if you want to try, the polygon edge walking code is written using macros which could easily be used directly inside the graphics layer.

The code you want to look at is in miwideline.c and miwideline.h

Although wide arcs have seen a substantial speedup since the original protocol-conformant code was shipped with R3, it would be nice they ran much faster. Unfortunately, the protocol defines an object which is quartic in description, the straight line code solution of which involves several square roots and a couple cube roots. If you examine the implementation in miarc.c, you'll see a nightmare of complicated floating point arithmetic. These routines attempt to come as close as possible to the protocol definition for a wide arc, and suffer tremendously in performance because of it. Wide circles, however, do provide a reasonable opportunity for optimization. As the both inner and outer edges of a wide circle are circular (unlike elliptical arcs), a fully-integer circle edge walker is used to scan-convert them. Zero width arcs (like lines) are not completely specified by the protocol, and so the traditional integer walker is included in mfb, cfb and mi. If you can't use one of those version directly, look in the relevant code in ddx/mi/mizerarc.c,mizerarc.h, ddx/mfb/mfbzerarc.c and ddx/cfb/cfbzerarc.c.

Also included in cfb for R5 are some interesting optimizations for clipping points to rectangles and rendering polygons. Look in ddx/cfb/cfb8line.c and ddx/cfb/cfbply1rct.c for these algorithms. The polygon code could easily be ported to mfb, however the zero-width line code is dependent on having addressable pixels.

PushPixels may also be an important routine to optimize. This is because it is used by the software cursor code. Both mfb and cfb have heavily tuned PushPixels routines which work in solid fill/copy mode and provide adequate performance for software cursors.