Godzilla’s Guide to Porting the X V11 Sample Server

March 1, 1988

Updated for R4: April 12, 1990

David S. H. Rosenthal
Sun Microsystems

Adam R. de Boor
University of California at Berkeley

edits by Bob Scheifler
Massachusetts Institute of Technology

Updated for R4 by Keith Packard
MIT X Consortium

ABSTRACT

For hackers in a hurry, here’s how to do a quick-and-dirty port of the X V11 sample server to memory-mapped monochrome & color frame buffers. The authors disclaim responsibility for damage to the environment during this process.

> ‘‘Did I – ?’’
>
> ‘‘I told you you could,’’ [Obi-Wan] Kenobi informed him with pleasure. ‘‘Once you start to trust your inner self, there’ll be no stopping you.’’
>
> Star Wars, *George Lucas*

---

1. Introduction

In two independent efforts, we ported the first alpha release of X V11 to monochrome Sun hardware. The results of these efforts were merged together, incorporated into the second alpha release, and delivered to MIT for distribution. We went on to port the second alpha release to a number of Sun 8-bit memory-mapped color displays, and we contributed this code to the beta releases.

With future releases, the internal structure of the server changed, and porting efforts have to be modified as well. No attempt has been made to isolate the porting efforts from the changes between various versions of the server. This document focuses on Release 4 of the MIT Sample Server.

Based on this experience, we present a guide for others attempting to port X V11. Our goal is to enable them to get the server up and functional as expeditiously as possible. We address only ‘‘initial’’ ports, using the *server/ddx/mfb*, *server/ddx/cfb* and *server/ddx/mi* code to drive a dumb memory frame buffer.

It is possible to port X V11 to dumb frame buffers very quickly, even without a detailed understanding of the internals of the server. We give instructions on how to do this, and then go on to cover some of the internals that will be important in doing a more complex port, and in performance tuning. We do *not* cover the following areas in detail:

- Porting to operating systems other than 4.2 or 4.3 BSD.
- Driving ‘‘intelligent’’ display hardware.
- Supporting input devices other than the mouse and keyboard.
- Performance tuning the system.

This document refers to the fourth full X V11 distribution from the MIT X Consortium.

2. Raw Materials

The server source code is organised into a number of directories:

ddx	The code for driving displays, keyboards, mice, and other peripherals.
dix	The machine independent code implementing the protocol interpreter and resource managers.
include	The *.h* files defining the machine independent data structures for the server.
os	The code implementing the server’s file and inter-process I/O, which depends on the operating system.

In the **ddx** directory, there are the following directories:

mfb	The ‘‘monochrome frame buffer’’ driver. This is intended to be a machine independent driver for 1-bit deep memory mapped displays.
mi	The ‘‘machine independent’’ driver. This is intended to be a machine independent (but slightly incomplete) driver for *any* frame buffer.
cfb	The ‘‘color frame buffer’’ driver. This is intended to be a machine independent driver for *n*-bit deep (*n* <= 32) Z-Format memory mapped displays. While much of this code is portable to arbitrary depth displays, release 4 provides a fast implementation only for the 8-bit case.
...	There are several directories which are specific to a particular vendor and contain minimal code for talking to that hardware. In the **os** directory, there is at present only the following directory:

The only subdirectory in the **os** directory is:

4.2bsd	An implementation of the file and interprocess primitives for the server using the facilities of 4.2 and 4.3 BSD. This code also works on various SYSV-BSD blends (A/UX, HPUX etc) but no attempt has been made to get it working in a generic SYSV environment.

The intent is that one should port the sample server to new hardware by using the *mi* code, implementing at first only the lowest level routines such as **FillSpans( )**. We did not do so for our initial monochrome ports (the *mi* code was too incomplete), instead we used the *mfb* code. Our subsequent color ports were based on *mi* and we describe both. The current release relies on *cfb* heavily as much of *mi* is only interesting from an academic viewpoint.

The details of server internals you need to know to do this, and more advanced ports, are described in a document (the *Porting Layer*) written by the DEC team that developed it.[2] You should read this document even for a simple monochrome or color port, a complete understanding isn’t important, but a better understanding of the basic porting process will result from even a cursory perusal. We refer to it frequently when more details are needed.

3. Road Map

In order to get the sample server running on your hardware, you will need to deal with four distinct areas:

a) Painting the Bits – you need to make the code work for the mapping between pixel coordinates and byte/bit addresses defined by your hardware.

b) Pounding the Keys – you need to map between the keystrokes and mouse movements your hardware provides, and the canonical forms required by the server.

c) Dragging the Cursor – you need to move a cursor image around the screen, and change the image on demand.

d) Starting Up & Shutting Down – you need to initialize and close down the connections between the server and your hardware.

4. Doing the Port (Monochrome and 8-bit color)

Lets suppose you want to port X V11 to the Generic Workstation Company’s (GWC) hardware. You can do most of the work in one fell swoop:

```
#!/bin/sh
cd server/ddx/sun
mkdir ../gwc
sed -e ’s/sun/gwc/g’ Makefile >../gwc/Makefile
for A in *.[hc]
do
    sed -e ’s/sun/gwc/g’ $A >../gwc/gwc‘expr $A : ’sun\(.*\)’‘
done
```

You now have a directory in the right place with a first approximation to the source files in it. It will avoid confusion if you remove all code in this directory between

```
#ifdef  SUN_WINDOWS
#endif  SUN_WINDOWS
```

This deals with running X V11 ‘‘on top of’’ the SunWindows window system, and is of no interest here.

The next steps tackle the four major areas in which GWC’s workstations are different from Sun’s.

4.1. Painting the Bits

Assuming that you have a machine which has a simple memory bitmap, all you have to do to ensure that the pixels are painted correctly is to deal with the potential differences between simple memory bitmaps. The files *server/include/servermd.h* and *fonts/bdftosnf/bdftosnf.h* contain the definitions for the various CPUs which already support the sample server. You will need to add #ifdef sections after the others in each file which contains the following definitions.

- Byte order. Both the *mfb* and *cfb* code can handle either byte order. If you have a 68000 CPU (Big Endian), the file *server/include/servermd.h* should contain:

  ```
  #define IMAGE_BYTE_ORDER MSBFirst
  ```

  If you have a 80386 GWC, *server/include/servermd.h* should contain:

---

[2] *The Porting Layer for the X V11 Sample Server*, by Susan Angebranndt, Raymond Drewry, Phil Karlton and Todd Newman. See the file *doc/Server/ddx.doc.tbl.ms*.

```
#define IMAGE_BYTE_ORDER LSBFirst
```

- Pixel order. Again, both the *mfb* and *cfb* code can handle either the 68000 style, where the most significant bit is to the left on the scan line, or the 80386 style, where the least significant bit is to the left on the scan line. For the 68000 GWC, the file *server/include/servermd.h* should contain:

  ```
  #define BITMAP_BIT_ORDER MSBFirst
  ```

  And the file *fonts/bdftosnf/bdftosnf.h* should contain:

  ```
  #define  DEFAULTBITORDER  MSBFirst
  ```

  For the 80386 GWC, the file *server/include/servermd.h* should contain:

  ```
  #define BITMAP_BIT_ORDER LSBFirst
  ```

  And the file *fonts/bdftosnf/bdftosnf.h* should contain:

  ```
  #define  DEFAULTBITORDER  LSBFirst
  ```

- Pixel meaning. Some machines have monochrome displays where a 1 bit means black, while others ones have displays where a 1 means white. Check the manual, and set the appropriate values in the file *server/ddx/gwc/gwcBW2.c*:

  ```
  pScreen->whitePixel = 0 or 1;
  pScreen->blackPixel = 1 or 0;
  ```

- Alignment restrictions. CPUs vary in the alignment restrictions they place on memory accesses. For example, some can do 4-byte accesses at every byte address, and some only at 4-byte boundaries. In general, the *mfb* and *cfb* code take the conservative viewpoint that the framebuffer and memory bitmaps are accessed only 4 bytes wide at 4-byte boundaries.

  However, this isn't true of the font code. For Sun hardware, we decided to extend the same restriction to the font code, and changed the padding rules used for the fonts to avoid those cases in which the *mfb* and *cfb* code would use addresses that weren't 4-byte aligned. We will cover the padding rules in some detail later; for the moment all you need to do is to include in the file *fonts/bdftosnf/bdftosnf.h*:

  ```
  #define  DEFAULTGLPAD 4
  ```

  while *server/include/servermd.h* should have

  ```
  #define GLYPHPADBYTES    4
  ```

The *mfb, cfb* and *mi* directories are now set up for your machine. The next steps are more work.

4.2. Pounding the Keys

More frequently than you would believe possible, the DIX layer will call the function **ProcessInputEvents( )**. This function lives in *gwcIo.c*, and you will be able to use it almost unchanged. It has to:

- Obtain the device-specific events from the keyboard and the mouse, by calling through their private descriptor structures to a device-specific **GetEvents** routine.
- In time-stamp order, hand each event to the appropriate **ProcessEvents** routine, again calling through the private descriptor structure for each device.
- Maintain the time of the last event.
- Restore the screen if it is currently saved.

Only the first of these requires modifications to the code you now have, and these take place in the files *gwcKbd.c* and *gwcMouse.c*

Each device has a **GetEvents** routine; it needs to return an array containing the device-specific events that are immediately available. The precise structure of the events is not important, but they need to be timestamped by the kernel. The declarations in *gwcIo.c*, *gwcKbd.c* and *gwcMouse.c* of things as pointers to **Firm_events** should be changed to **gwc_events**, the name of the structures the GWC kernel returns.

To obtain the events, you can either do a non-blocking **read( )** or use a shared-memory circular queue of events, if your kernel supports it. The *server/ddx/sun* code supports only non-blocking reads, since current Sun kernels do not have a shared-memory event queue. You are strongly urged to add support for a shared-memory event queue to the GWC kernel, since doing the non-blocking **read( )**s is a serious performance problem.

For the present, we assume that the GWC kernel has separate */dev/mouse* and */dev/kbd* files, which:

- Support the **fcntl(FNDELAY)** call.
- Provide a stream of **gwc_event** structures containing position, keycode, and timestamp information.

In *gwcMouse.c*, you will need to change only the names of the fields in the device-specific event structures, from those of the *Firm_event* to those of the *gwc_event*, and the values of the ID codes (such as MS_LEFT) used for the button and motion events. If you have one of the GWC mice that reports absolute positions rather than X and Y deltas, you will need to add another case to the following statement.

```
switch (fe->id) {
    case MS_LEFT:
    case MS_MIDDLE:
    case MS_RIGHT:
    case LOC_X_DELTA:
    case LOC_Y_DELTA:
    default:
}
```

Beware of the Sun convention that motion up gives a positive Y delta.

In *gwcKbd.c*, you will need to make similar changes to the names of the event fields. Then, replace the various **ioctl(KIOC<foo>)** calls with their GWC equivalents Their meanings are:

Name	Operation
KIOCTYPE	Get int indicating keyboard type
KIOCGTRANS	Get/set keycode translation. We want ASCII events.
KIOCSDIRECT	Switch keystrokes between */dev/kbd* and */dev/console*. We want */dev/kbd*.

Now, you have to establish a mapping between the codes your keyboard sends and the names X V11 uses for keys (the so-called *keysyms*). Read the files *server/ddx/gwc/gwcKeyMap.c* and *X11/keysym.h*. Look at your keyboard manual, and find the lowest keycode it sends (*kcmin*) and the highest keycode it sends (*kcmax*). Look at the keyboard itself, and find the key with the largest number of symbols on it, counting strings like ‘‘Return’’ or ‘‘F9’’ as one symbol. The number of symbols on this key is the *width* of the keyboard. Now, for each of your keyboard types, you will need two things in *gwcKeyMap.c*:

- A *keymap*, which is an array with (*kcmax*-*kcmin*+1) rows and *width* columns. Each row in the array should contain the keysyms corresponding to the symbols on the keycap, with the first column containing the symbol generated when the key is un-shifted, the second the symbol generated when the key is shifted, and the other columns the other symbols in no special order. So, for example, a key with the symbols ‘‘1’’ and ‘‘!’’ on it would have the keysyms *XK_1* and *XK_exclam* in the table.
- A *modifier map*, which is an array with one entry per keycode. Each entry should contain the modifier bits that are set when that key is down. So, for example, a ‘‘Shift’’ key would have an entry *ShiftMask*.

4.3. Dragging the (Software) Cursor

Suns use a software cursor, as there is no cursor hardware. The problem with is that it must be removed from the display before painting operations that might affect the pixels it is using. And, of course, put back again at some time later.

Fortunately, a machine-independent software cursor layer is included in *server/ddx/mi* which is easily connected to the mouse device. It will work on any display, but it is a little slow, and you will want to use the hardware cursor on machines that have it. The changes needed to do so are fairly extensive, so we cover them later also.

4.4. Starting Up & Shutting Down

When the server is started, it initializes its output and input devices by calling their initialization procedures. This is where things get really device-specific, and the code for the GWC will differ significantly from the Sun code. The overall structure will be preserved, however.

It is important to observe that, once the last client has closed its connection and everything has been shut down, the server will re-initialize everything by repeating the process. Although there is a close-down procedure, it is generally better to avoid closing the device, instead simply reset it to its initial state. So if, for example, initializing a display or a mouse involves opening a file, the descriptor should be remembered in a static structure and not re-opened if it is already open.[4]

4.4.1. Output

Output devices are initialized in a two-step process:

- The server calls **InitOutput( )**, a routine in *gwcInit.c* which you can re-use untouched. It calls each potentially available display's probe routine, finding the probe routine and the probable file name for it in the **gwcFbData[ ]** array.

  The dumb monochrome device, whose driver is in *gwcBW2.c*, has a probe routine called **gwcBW2Probe( )**. This has to attempt to initialize the monochrome display and, if it succeeds, fill out the **fbFd** structure describing it. In the Sun case, most of the work is done in a routine **sunOpenFrameBuffer( )** in *sunInit.c*, because it is common among all Sun framebuffers. This routine scans the command-line arguments, the environment, and the */dev* directory to find a frame-buffer of the required type, opens it, and returns the file descriptor.

  The dumb color device, whose driver is in *gwcCG3.c*, is similarly structured, the probe is called *gwcCG3Probe( )*.

  The probe routines then have to map this descriptor into the server's address space, and install a pointer to the pixels in the **fbFd** structure. Note that the Sun monochrome code has to deal with two different sizes of monchrome framebuffers. The **fbFd** structure is the static structure we mentioned earlier, needed to preserve internal driver information across server re-initializations. Finally, it calls **AddScreen( )**, giving it the address of the BW2/CG3 initialization routine.

- **AddScreen( )** fills out the screen information, and calls the initialization routine. In the monochrome case, **mfbScreenInit( )** is called to initialize the *mfb* part of the code. This fills in the **Screen**'s operations vector with the routines the *mfb* code supports. The remaining entries are filled out by the initialization routine itself. This code can be reused unchanged. The color case simply uses **cfbScreenInit( )**. Note that in the color case the values for whitePixel and blackPixel are left unspecified. *cfb* will fill in the values 0 for white and 1 for black. If this is not satisfactory, replace them with your own values after calling **cfbScreenInit** and before calling **cfbCreateDefColormap**.

When the server is shutting down prior to re-initialization, it will call the **CloseScreen** function in the **Screen** structure. You can also use this routine unchanged.

One other area which needs attention are the **gwcBW2SaveScreen( )** and **gwcCG3SaveScreen( )** routines. These need to enable and disable the video for the GWC framebuffer.

---

[4] See section 2.4.3 of the *Porting Layer*.

4.4.2. Input

A similar two-step process is used to initialize the input devices. First, the server calls **InitInput( )**, in *gwcInit.c*. It registers the keyboard and mouse devices by calling **AddInputDevice( )**, among the arguments to which are the appropriate initialization/closedown routines **gwcMouseProc** and **gwcKbdProc**.

Then, the server calls each of the device initialization/closedown routines twice, once with command **DEVICE_INIT** and once with command **DEVICE_ON**. These must:

- Open the necessary devices.
- Set up the appropriate keymaps.

When the server is shutting down prior to re-initialization, it will call the device's initialization/closedown routine with command **DEVICE_OFF**.

Except for the details of how the devices are opened and coerced to supply ASCII events, and the changes made earlier to the **ioctl( )**s, the rest of this code can be used unchanged.

4.5. Tidying Up

You have now made all the major changes needed. All that remains is to make suitable changes to the makefiles:

- In *server/Imakefile*, make the following changes:

```
     ALLDDXDIRS = whatever is already there ddx/gwc
        GWC = ddx/gwc/libgwc.a
    ALLPOSSIBLE = whatever is already there Xgwc

#ifndef XgwcServer
#define XgwcServer /* as nothing */
#endif
            ALL = whatever is already there XgwcServer

#
# GWC server
#
GWCDIRS = dix ddx/snf ddx/mi ddx/mfb ddx/cfb ddx/gwc os/4.2bsd
GWCOBJS = ddx/gwc/gwcInit.o
GWCLIBS = $(GWC) $(CFB) $(DIX) $(SNF) $(UNIX) $(MFB) $(MI) $(EXTENSIONS)
GWCSYSLIBS = $(SYSLIBS)
XgwcDIRS = $(GWCDIRS)

ServerTarget(Xgwc,$(EXTDIR) $(GWCDIRS),$(GWCOBJS),$(GWCLIBS),$(GWCSYSLIBS))
```

Figure 1: Top-level Imakefile Changes

- Now go ahead, make everything, and enjoy!

There are probably some details we've omitted – we no longer have our GWCs so we're writing this from memory. If you find either the details or the hardware, please let us know.

5. Additional work needed for doing a Color Port on non 8-bit displays.

After the *mfb*-based monochrome drivers were shipped to MIT, we went on to develop drivers for Sun's color hardware. Our initial attempt was based on the *mi* code. Our goals were:

- To get the server functional on color hardware as expeditiously as possible.
- To test the color code in *server/dix*, and as much as possible of the code in *server/ddx/mi*.
- To test the recommended porting strategy, using *mi* and implementing only **GetSpans( )**, **SetSpans( )** and **FillSpans( )**.

- To provide a highly portable implementation of the DDX layer for color hardware to enable others to get the server running as effortlessly as possible.

Performance was explicitly not a goal. Just as *mfb* assumes that the display it is driving has a 1-bit deep memory framebuffer accessed 32 bits wide at 32-bit boundaries, *cfb* assumes a *n*-bit deep ($n <= 32$) memory framebuffer accessed 32 bits wide at 32-bit boundaries. While these restrictions are fairly onerous, they make the code highly portable in both cases. In the monochrome case, the DEC team managed to provide relatively good performance. In the color case, there is *n* times more work to do, and we have not attempted all of the optimizations that *mfb* uses.

5.1. Porting cfb to non-8 bit framebuffers

By default, *cfb* is set up to drive a framebuffer that is 8 bits deep with the pixel order defined by **BITMAP_BIT_ORDER**. If your framebuffer is different, you will need to change some parameters in *server/ddx/cfb/cfbmskbits.h*:

Parameters in cfb			
Parameter	mfb	cfb	Comment
PPW	32	4	pixels per word
PLST	31	3	last pixel in a word (should be PPW-1)
PIM	0x1f	0x03	pixel index mask (index within a word)
PWSH	5	2	pixel-to-word shift
PSZ	1	8	pixel size (bits)
PMSK	0x01	0xFF	single-pixel mask

Read the comments in *cfbmskbits.[hc]* carefully before changing these parameters. You will also need to change the mask values in *server/ddx/cfb/cfbmskbits.c*.

Changing these parameters and masks is all you should need, but we cannot be sure. The code has been used on several types of color hardware, but in each case there are four pixels to the word (and, therefore, the masks don't need changing).

5.2. Using cfb

The code in *server/ddx/sun/sunCG4C.c* illustrates how to use *cfb* to drive an 8-bit deep memory frambuffer (the usage on non 8-bit deep frame buffers is identcal but no example code exists). You call **cfbscrinit( )** to fill out the screen operation vector, and **cfbCreateGC( )** to create a GC. In both cases, reading the code will show that most of the procedures to be used are from *mi*.

Unlike *mfb*, which effectively supports only a StaticGray visual, the *cfb* code supports all 6 possible visual types if your hardware has writeable color maps. Otherwise, (or for testing purposes) you can define **STATIC_COLOR** and support a StaticColor visual (which also limits the supported visual types to StaticColor and TrueColor).

6. Details

This section is not intended to be a complete survey of the details of the server. We wouldn't claim enough knowledge to write that (yet). It is rather a collection of comments on the areas we have had to deal with in detail, in the hope that we can save others from wasting their time on problems that are either already solved or insoluble.

6.1. Software & Hardware Cursors

The details of the DIX interface to the cursor support are described in section 2.5 of the *Porting Layer*.

6.1.1. Software Cursors

The Machine Independent software cursor code is very easy to use and it provides reasonable performance. It works by "wrapping" every potential drawing operation to the screen and checking for overlap with the cursor during these operations. The *Porting Layer* describes this "wrapping" notion in greater detail.

6.1.2. Hardware Cursors

To switch to using a hardware cursor, you will need to study the QVSS code,[12] and to remove the following code from the *server/ddx/sun* files:

- *sunInit.c*: the code that calls miDCInitialize
- *sunMouse.c*: the code in **sunMouseProcessEvent( )** that positions the software cursor.

6.2. Fonts & Padding

The *Porting Layer* describes the layout of the glyph information in memory as follows:[13]

> ‘‘Each scanline of each glyph is padded to a byte boundary with zero bits. Bit and byte order is whatever is natural for the server. (Note: the current BDF to SNF font compiler handles either bit order within a byte as a compile time option. It does not deal with byte order.) The glyph for a character whose XCHARINFO is ci begins at cg[ci.byteOffset]. Glyphs may begin at arbitrary offsets within the array.’’

The section on ‘‘Alignment’’ describes the problem:[14]

> ‘‘The mfb text code might access mis-aligned longwords; this is not a problem on VAXes, 680x0 (x != 0), or Intel architectures, but might be on some as yet unknown processors, and is definitely bad on a 68000. An easy fix is to have the font compiler generate longword padded glyphs instead of byte-padded ones.’’

We followed this advice, and the results are incorporated in the Sun code in the release. The trade-offs to consider when deciding what to do are:

- Many processors cannot access mis-aligned longwords.[15] In these cases, there is no real choice.
- Even processors which can access longwords at byte boundaries do so more slowly than aligned accesses, and these accesses are made in a performance-critical area (painting characters). Check your hardware manual, or write a small benchmark. Furthermore, the font code for big-endian machines is always slower when using non-longword padded fonts as it must shift things around.
- On the other hand, padding the glyphs to longwords wastes a significant amount of space.

If you’d like to pack fonts even on machines which can’t access longwords on non-longword boundaries, then, in the file *server/include/servermd.h*, choose the alignment restriction (in bytes) for those accesses:

```
#define GETLEFTBITS_ALIGNMENT1   /* arbitrary byte alignment */
#define GETLEFTBITS_ALIGNMENT2   /* halfword alignment */
#define GETLEFTBITS_ALIGNMENT4   /* longword alignment */
```

If you are using GLYPHPADBYTES 4, make sure you set GETLEFTBITS_ALIGNMENT to 1 even if your CPU can’t access longwords on non-longword boundaries. This is because the text code would use extra instructions to check the alignment of the accesses even though the fonts are guaranteed to be longword aligned.

6.3. Shared Event Queue

By default, the DIX layer calls **ProcessInputEvents( )** before it it waits for something to happen, and also before performing each client request. When using non-blocking **read( )** calls this is a substantial overhead, and a facility for reducing this has been provided. **SetInputCheck( )** can be called with the addresses of two locations, and **ProcessInputEvents( )** will only be called when they differ. There are two ways of using this:

- If your mouse and keyboard drivers support **SIGIO**, enable this mode when you open them, and register a handler that increments a location. Give **SetInputCheck( )** the address of this and a zero location,

---

[12] See the files *server/ddx/sun/sunCursor.c* and *server/ddx/dec/qvss/qvss_io.c*, and section 2.5 of the *Porting Layer*.

[13] See section 5.3.

[14] See section 4.5.2.

[15] Try it on a 68010 sometime, or on a PC/RT, or on a SPARC, or ....

and add code to re-zero the location to **ProcessInputEvents( )**. The Sun code does this.

- If you have a kernel event queue that can be mapped into a user process, give **SetInputCheck( )** the addresses of the head and tail pointers.[17] This is the best alternative, because it eliminates the **read( )** system calls as well.

In both cases, there is an interaction with the software cursor code. When the cursor is out of the bitmap, this mechanism must be disabled. We need to ensure that **ProcessInputEvents( )** will be called at some time soon after the cursor is removed in order to put it back.

- If you are using **SIGIO**, you can simply call the SIGIO handler from the cursor removal code.
- If you are using a shared event queue, you have to call **SetEventCheck( )** in the cursor removal code, giving it the addresses of two locations that are always different, and again in the code that paints the cursor, giving it the head and tail pointers.

6.4. Multiple Screens

The *sun* code supports multiple screens (‘‘Zaphod’’ mode).[18] There are two possible versions of Zaphod mode:

- ‘‘Passive’’, in which the cursor stays on one screen until some client actively warps it to another screen.

  ‘‘Active’’, in which the server warps the cursor between screens itself.

The *wm* window manager implements passize Zaphod mode, clicking on the background warps the pointer to the next screen in sequence.

The *sun* code implements active Zaphod mode by using the mi Cursor code. In **sunCursorOffScreen( )**, when the pointer gets to the right (left) edge of the current screen, it is warped to the next (previous) screen in sequence. This is done by simply modifying the input paramenters and returning the appropriate value. If you aren’t using the mi software cursor code, examine it for the details on how to get the cursor to the new screen.

7. Conclusions

During the alpha & beta testing periods of X V11, many interpid porters attempted to adapt the sample server to their hardware. It is to their credit, and especially to the spirit of cooperation in which the implementors received the enormous volume of comments and suggestions that resulted, that the X V11 sample server is now remarkably easy to get running on new hardware. A few bold strokes of the keyboard, and the task is done. Well, almost, but not quite. What remains is to tune the server to give of its best on your particular hardware. We leave this as an exercise for the reader.

8. Acknowledgements

Thanks to John Ousterhout and Andrew Cherenson for various suggestions in the initial UCB port.

The Sun port of the alpha server was done by David Rosenthal, Mike Schwartz, Stuart Marks, Robin Schaufler, and Alok Singhania. It was made much easier by the extent to which we could steal from the Sun X.10 server, now the result of too many people’s work to acknowledge individually. However, Paul Borman of Cray Research did particularly useful work on keyboard support.

The *cfb* driver was originally the work of Stuart Marks (from a vague idea by David Rosenthal), with help from Jack Palevitch (now at Apple) and Bob Leichner of H-P Labs. The current *cfb* release was written at the MIT X Consortium for release 4 and shares a few file names in common with the original code.

The version of the Sun code in the MIT release is the work of Adam de Boor, David Rosenthal, Stuart Marks, Robin Schaufler, Frances Ho, Mark Opperman and Geoff Lee. Integration of the Sun code into the MIT release would have been impossible without the generous help of the Statistics Center at MIT, who allowed us to monopolise their Suns at all hours of the day and night, and Todd Brunhoff of Tektronix.

---

[17] See the file *server/ddx/dec/qvss/qvss_io.c*.

[18] See *The Hitch-Hiker’s Guide to the Galaxy* by Douglas Adams.

Special thanks to the Generic Workstation Company for the loan of their hardware.