Porting an X11R4 Server to the XINU Operating System

Steve J. Chapin    Douglas Comer    James Griffioen    Scott Mark
Patrick Muckelbauer    Shawn Ostermann

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract
In this paper we discuss a port of the X11R4 sample server to the XINU operating system running on both CISC and RISC architectures. What makes these ports unusual is our freedom to define or alter the underlying operating system. The paper focuses on three aspects of the port: the advantages of using a relatively new and dynamic base environment, development of a UNIX system call emulation library to ease porting, and modifications to the device-dependent layer of the X server that allow easy portability between different machine architectures.

---

*This work was supported in part by Digital Equipment Corporation grant number N504 and National Science Foundation grant number ASC-851-5851.

†XINU is a compact system that provides UNIX-like kernel services, multi-threaded processes, a virtual memory system, IPC, and internetworking.
1 Introduction

1.1 Project Goals

This paper describes a transport of the X server\footnote{Throughout this paper, the terms X server and server refer to the sample server from X11 release 4 as distributed by MIT.} to the Xinu operating system. Our goal is to design minimal operating system support that provides services needed to run the X server on a high-speed RISC architecture computer. We have adopted the following constraints to make the resulting system usable in a diskless configuration in a conventional network setting:

- A small amount of memory (4 total megabytes of RAM and ROM or less)
- Minimal bootstrapping support from the network
- Remote file access for font files
- Remote paging store for virtual memory
- Standard network technology (the prototype uses Ethernet)
- Standard network protocols (the prototype uses TCP/IP protocols)

Finally, to make maintenance easy and allow upgrades of the server, we decided to pursue a design that kept changes to the UNIX version of the X11 server software to a minimum.

1.2 Project Design and Overview

Although Xinu possesses many of the characteristics of UNIX, it is not compatible with UNIX, nor is it intended as a replacement of UNIX in a production environment. Some system calls are similar (e.g., `read()` and `write()`), but others differ in both syntax and semantics. Xinu was designed to codify the powerful abstractions provided in other operating systems in a clean and elegant way, while maintaining a small kernel. It is primarily intended as a research vehicle and therefore does not have many of the amenities to which UNIX users are accustomed. Xinu does provide processes, semaphores, inter-process communication, network communication, internetworking support, and virtual memory.

The chief advantage of working with Xinu comes from its flexibility. We started with a clean slate and were free to define the operating system interface for the mouse, keyboard and frame buffer. Drawing from a base of knowledge and experience with windowing systems that was not available a decade ago, we chose abstractions for device drivers that provided a more versatile and uniform interface than was available across various UNIX implementations.

Figure 1 shows the final structure of system. The UNIX emulation layer consists of a library of routines that execute outside the operating system kernel.

The remainder of this paper describes the work done to port the X server, and is divided into four sections. Section 2 describes some of the facilities provided by Xinu. Section 3 discusses the services required by the server and how Xinu provides them, including services found in the kernel, a UNIX emulation layer written to map UNIX system calls to Xinu calls, and abstract devices that facilitate porting complex applications. Finally, section 4 draws conclusions and considers improvements and further developments of the system.
2 **XINU Facilities**

XINU, while not designed with the intention of being compatible with UNIX, shares some of its functionality. Although the interfaces often differ between the two systems, the steps necessary to emulate UNIX system calls in XINU are fairly straightforward in the case of most common operations. In this section, we discuss those aspects of the XINU operating system that differ significantly from their counterparts in UNIX. Operations making use of these facilities for UNIX-like services require more effort to emulate. The most interesting of these facilities are XINU’s process model, its network interfaces, and its remote file system.

2.1 **Virtual Memory**

The XINU virtual memory system differs from that of UNIX in several important ways. The major differences are in the areas of process and thread creation, address space sharing, and backing store.

2.1.1 **Address Spaces**

XINU supports a more powerful process model than conventional UNIX implementations. It supports multi-threaded user processes allowing concurrent manipulation of shared data within an address space. A thread is defined as a point of execution within an address space, along with its associated state information. All threads within the same address space execute instructions from the same text region, each at an independent point in the code. In addition, the data and bss regions of an address space are shared by all threads executing within it. Semaphores and efficient inter-process communication provide synchronization between threads, whether or not they are in the same address space.

The XINU kernel consists of a single address space encompassing multiple, independent kernel threads. Each kernel thread corresponds to a separate task in the system, and can communicate
with its peers directly using shared memory IPC. This multi-threaded subdivision of labor simplifies the design of kernel tasks such as page reclamation, network management, background paging, etc.

2.1.2 Process Creation

XINU supports three types of process creation: a new thread created in the kernel address space, a new thread created in an existing user address space, or a new thread created in a new address space. This added flexibility complicates the issue of emulating UNIX style process creation. XINU does not have a notion that directly translates into a UNIX \textit{fork()} call, although it does have an operation similar to the UNIX \text{exec()} system call which starts a new user process from an executable image on disk.

2.1.3 Remote Memory Model

When user programs exhaust the local physical memory, an operating system writes blocks of physical memory to some manner of backing store, generally magnetic disks. Magnetic disks provide high data transfer rates, large storage capacity, and the ability to randomly access data, making disks an appealing backing storage media. The operating system typically reserves a fixed size section of the disk for backing storage and writes blocks of data directly to that section of the disk. The virtual memory system later retrieves the blocks of memory from the disk on demand.

More recent virtual memory systems have added a level of abstraction to the paging paradigm. These systems use the abstraction of files to hide the underlying disk device from the virtual memory system allowing the operating system to store data on the disk using high level file operations. The virtual memory system does not need to know the characteristics or layout of the underlying disk device because the file system handles the transfer of data to the disk. Some operating systems provide support for a distributed file system, using remote files for backing storage. Unfortunately, the file abstraction increases the overhead associated with paging. Writing data to a file usually requires a minimum of 2 disk accesses, 1 or more to update the directory structure and 1 to write the data. Moreover, file systems often use read-ahead techniques to improve performance of sequential accesses. They can exhibit poor performance when used to store pages from virtual memory because paging activity generates random fetch and store requests.

XINU uses a new model of virtual memory in which dedicated, large-memory machines serve as backing storage for virtual memory systems operating on a set of clients. The dedicated memory server allows sharing of the large physical memory resource and provides fast access to data. Figure 2 illustrates this model.

2.2 File System

The existing XINU system provides permanent data storage through the use of a remote file paradigm. The XINU Remote File System utilizes dedicated file servers running on remote machines, and a specially designed file access protocol. This allows XINU full access to a wide variety of files, but keeps the stand-alone XINU kernel small. To simplify the task of integrating XINU into an existing environment, we are also working to provide NFS device drivers in the XINU kernel.

2.3 Namespace

An important service provided by an operating system is the mapping of names to physical devices. In UNIX, this mapping is done almost exclusively through its hierarchical file system. UNIX specifies the interpretation of file path names and breaks them into directory and file names separated by
slashes. In order to access most UNIX devices, for example, the user opens a special pseudo-file in the “/dev” directory.

In XINU, this type of mapping is left to the user and is accomplished through a syntactic name space. The NAMESPACE device contains a table of name prefixes to match, each with a corresponding new prefix and new device. When the NAMESPACE device is opened with a name, XINU locates the first matching prefix in this table and replaces the corresponding prefix in the name string. The new name is then passed to a recursive open on the new device also specified in the table. This recursion continues until the device is no longer NAMESPACE. For example, the table in figure 3 emulates a UNIX-like hierarchical file system.

When the user opens the file “/bin/X” on a NAMESPACE device configured as above, the device applies the next to last transformation (the first exact match), replacing the “/” with “XINU/”, and performing a recursive open of “XINU/bin/X” on the NAMESPACE device. This second open is transformed using the “XINU/” rule, and becomes an open of “/usr/sdo/vmixinu/bin/X” on the RFILSYS device. This finally results in the opening of a physical file on the file server. Many
"/dev/Xlog" -> (RFILSYS) 
"/dev/fb" -> (CFB) 
"/dev/debug1" -> (NULLDEV) 
"/dev/debug2" -> (NAMESPACE) 

Figure 4: sample NSPACE rules

interesting alternatives are possible; consider the rules shown in figure 4.

The first rule maps the name “/dev/Xlog” to the specified remote file, “/usr/sdo/Xlog”. The second rule is used to map the name “/dev/fb” to either the color frame buffer, CFB, or the monochrome frame buffer, MFB. The next two NSPACE mappings allow a program to log debugging information at two levels, debug1 and debug2. Depending on the entries in the NSPACE, these messages are either discarded (as they are for “/dev/debug1”) or logged to a remote file (as for “/dev/debug2”).

2.4 Internetworking

XNU implements both the UDP and TCP protocols from the TCP/IP protocol suite. The interfaces to these routines were designed to be easy to use in the most common cases. For either protocol, a user can open a network connection by opening the appropriate device together with a hostname:port pair, as in “arthur.cs.purdue.edu:23”. Control calls are provided for tasks such as specifying the maximum size of the TCP listen queue, determining whether or not to use UDP checksums, and accepting listen-state TCP connections. We believe that notions such as the UNIX socket abstraction belong at a layer outside the kernel; the feasibility of this view is clearly demonstrated in section 3.3.2 which describes the UNIX emulation library.

3 Supporting the X server

This section discusses the work done to support the X11R4 sample server on the XNU operating system. We first examine the operating system services required by the server. The next subsection covers the portions of those requirements provided by the XNU kernel, and then we describe software that provides a UNIX emulation layer outside of the kernel. The last portion describes abstract devices and their impact on porting the server between disparate hardware platforms.

3.1 Operating System Requirements

The X11R4 server is a large, complex piece of software, and demands many operating systems services. Chief among these is the ability to communicate with client processes over a reliable byte stream. A reliable byte stream delivers all data, in order, exactly once. In most implementations, this means either a connection oriented, streaming network protocol such as TCP for remote clients or a pipe mechanism for local clients. XNU uses the TCP transport protocol for both local and remote clients.

The X server requires a file system for storage of fonts, color maps, authorization files, log files, and other miscellaneous data. XNU provides a remote file system that allows a diskless workstation running the X server to access files on a remote machine. The server also needs several operating system primitives, including process creation, execution of external programs,
int
xread( int device , char * buffer , int length )

Blocking:
    Read length characters from device.
Nonblocking:
    Read as many characters as are available, up to
    length, from device without blocking.
Return Values:
    BLOCKERR     nonblocking mode and 0 characters read
    0 < n <= length    n, the number of characters read

Figure 5: pseudocode for xread()

synchronous I/O multiplexing, control of system resource allocation, process synchronization, and
inter-process communication.

A minimum server configuration includes two input devices (a keyboard and a pointer device)
and one output device (a memory-mapped frame buffer). Additional input devices and frame
buffers may be used.

3.2 Kernel Support

One goal of an operating system is to provide an abstraction of the underlying hardware devices.
This relieves the user of dealing with complicated devices directly and permits a uniform inter-
face across disparate architectures. XINU provides this abstraction by defining a set of abstract
operations on devices. The operations include xread(), xwrite(), xgetc(), xputc(), and xcontrol().

Loosely speaking, xgetc() and xputc() deal with the transfer of a single character from of to
the device. The xread() and xwrite() operations function like xgetc() and xputc() but operate on a
contiguous block of memory. The xcontrol() operation allows the user to modify the device driver
parameters.

Most devices can run in one of two modes, blocking or nonblocking. In blocking mode, the
xread() and xwrite() operations complete the entire request before returning, blocking if necessary.
In nonblocking mode, the xread() and xwrite() operations fulfill as much of the request as possible
without blocking. This may require truncating the request. In either case, the return codes for the
xread() and xwrite() operations indicate the number of characters actually transferred. Switching
between blocking and nonblocking mode is handled through the xcontrol() operation. Pseudocode
for xread() and xwrite() is in figures 5 and 6.

XINU provides for synchronous I/O multiplexing through its zselect() system call. zselect() takes
as its arguments a bitmask of devices to monitor for input, a bitmask of devices to monitor
for output, two additional bitmasks for returning results, and a timeout. The timeout represents
how long a process is willing to block waiting for I/O, where a negative value indicates that the
process can block indefinitely. The zselect() call returns a code indicating whether the call timed
out or was successful. In the case of a successful return, the two result bitmasks are set indicating
which devices are ready for I/O.

The zselect() call registers itself with each input device of interest, requesting to be notified
when input becomes available. Registration takes two arguments, the process id to notify and a
bitmask. When input becomes available on the device, it sets its bit in the bitmask and notifies
int xwrite( int device , char * buffer , int length )
   Blocking:  
       Write length characters to device.
   Nonblocking:  
       Write as many characters as space available, up to
       length, to device without blocking.
Return Values:  
   BLOCKERR nonblocking mode and 0 characters written
   0 < n <= length   n, the number of characters written

Figure 6: pseudocode for xwrite()

the process. In a similar fashion, xselect() registers itself with each requested output device. After
registration, the process calling xselect() blocks waiting for notification from one of the devices or
for the timer to expire.

Notification is handled using XINU’s asynchronous inter-process communication, or IPC, mech-
anism. XINU provides three primitives for IPC, xsend(), xreceive(), and xrecvtime(). xsend() takes a
message and a process id as arguments and delivers the message to the specified process. xreceive() waits
for a message to arrive and then returns it to the caller; it requires no arguments. xrecvtime() takes a
timeout as an argument. If no message arrives within this prescribed time, xrecvtime() returns a special
error code, TIMEOUT. Pseudocode for xselect() is in figure 7.

3.3 Library Support

To make our port compatible with new releases of X, we decided to minimize the changes to the
server code. Our design uses a UNIX emulation library to achieve compatibility; this library maps
a subset of the UNIX system calls to corresponding XINU system calls. If the server makes a call
outside the supported subset, our library returns an error code indicating failure. The emulation
library allows us to use the standard UNIX C library, further reducing the number of changes to
the X server.

Building a UNIX compatibility library required us to add new functionality to the XINU kernel.
In particular, XINU did not uniformly provide asynchronous I/O, nor did its I/O system support
a disjunctive blocking facility similar to UNIX select(). While the multi-threaded capability in
XINU provides a more abstract alternative to such facilities, our decision to minimize changes in X
dictated the use of a single-threaded server.

3.3.1 The Device Table

Central to the emulation of UNIX calls is a mechanism to map from UNIX file descriptors and
their associated information to XINU device descriptors and their associated information. We
implemented an array in the per-process user address space called the device table (dtable). The
type of an entry in the device table is shown in figure 8.

The inuse field is a boolean that specifies the state of the device table entry; it contains TRUE
if and only if the entry is allocated. The remaining fields in the entry are not valid unless inuse
has the value TRUE.

8
int xselect( bitmask input_devices, bitmask output_devices,  
            bitmask input_devices_ready, bitmask output_devices_ready,  
            long timeout )
{
    for each bit i set in input_devices do
        registerForInput( i, process id,  
                            bitmask input_devices_ready )

    for each bit i set in outputs do
        registerForOutput( i, process id,  
                           bitmask output_devices_ready )

    if ( timeout < 0 ) then ret = xreceive()
    else ret = xrecvtim( timeout )

    for each bit i set in inputs do
        unregisterForInput( i )

    for each bit i set in outputs do
        unregisterForOutput( i )

    if ( ret == TIMEOUT ) then return TIMEOUT;

    return OK;
}

Figure 7: pseudocode for xselect()

struct devicetable {  /* per process file descriptor table */
    int inuse;  /* bool: true if entry in use */
    int xinudevnum;  /* corresponding xinu device */
    int misc_type;  /* type of misc pointer */
    void *misc_ptr;  /* pointer to misc. info structure */
};

Figure 8: the devicetable structure
struct UnixsockStr {
    int family;        /* protocol family (INET, etc.) */
    int type;          /* SOCK_STREAM, etc. */
    struct sockaddr name; /* address of peer */
    int namelen;       /* length of sockaddr */
    int state;         /* unbound, listen, etc. */
};

Figure 9: The UnixsockStr structure

xinnu.devnum contains the XINU device number corresponding to the UNIX file descriptor number. The misc_type field is a tag field whose value determines the interpretation of the miscellaneous pointer field. The miscptr field is an untyped pointer to an external data structure used to hold additional information. This information depends on the kind of UNIX object to which the entry corresponds, e.g. a socket or a conventional file.

An example use of these fields can be found in the XINU emulation of the UNIX socket mechanism and its associated system calls, specifically socket(), bind(), listen(), accept(), and connect().

3.3.2 UNIX Socket Emulation

UNIX includes the notion of a generic socket which is an endpoint for communication. The programmer allocates a socket via the socket() call, then incrementally provides more information about its use through the bind(), listen(), accept() and connect() calls. These calls allow the programmer to specify the other endpoint for communication, set the number of outstanding connection requests to allow, accept an incoming connection, and connect to a possibly remote application, respectively.

There is a fundamental difference in the semantics of data communications primitives in XINU and UNIX. As described above, the socket concept allows for late binding of a connection. In XINU, the full complement of information is supplied at the time the communication endpoint is allocated. Thus, knowledge must be accumulated through the successive socket calls until enough information is available to make the appropriate XINU call. This is accomplished through the use of the miscptr field in the dtable structure. For the socket emulation calls, the value DVT_SOCK is placed in misc_type, and miscptr is assigned the address of a UnixsockStr structure (see figure 9).

The socket() call reserves a slot in the device table, allocates a UnixsockStr, sets the misc_type field to DVT_SOCK, and assigns the address of the structure into the miscptr field in the device structure. The family and type fields in the socket structure depend upon the arguments passed to socket(). The state of the socket is set to UNBOUND.

bind() fills in the name field with the name of the local endpoint. In the case of TCP connections, this is an internet address and a port number. bind() only works on sockets in the UNBOUND state, and sets the state to BOUND.

Once a socket is bound, the programmer can either actively open a connection, which attempts to connect to a remote process, or perform a passive open, which waits for a connection from a remote process. In the case of an active open, connect() takes a socket in either the BOUND or UNBOUND state, sets the state to CONNECT, and opens the connection with the XINU zopen() call. If the socket was in the UNBOUND state, an implicit bind is done.

To perform a passive open, the programmer must first declare the maximum number of outstanding connection requests allowed for the socket. This is done with the listen() call. The socket
must be in the BOUND state, and its state is changed to LISTEN. The XINU device is opened at this point, and a `xcontrol()` call is used to set the queue size. Once this queue size is set, the `accept()` call waits for an incoming connection request. It also is implemented as a XINU `xcontrol()` call, which returns a new socket identifier that is connected and ready to be used for communication. The original socket is still in the LISTEN state and may be passed to `accept()` again.

The state transitions for a XINU socket are detailed in figure 10. The nodes on the graph are labeled with the state of the socket, and the arcs are labelled with the UNIX/XINU system calls that move sockets between states. It should be noted that the `accept()`/`xcontrol()` arc generates a new socket and leaves the original one with its state unchanged.

### 3.3.3 Synchronous I/O Multiplexing

UNIX achieves synchronous I/O multiplexing through the `select()` system call, which returns when one of the requested devices is ready for input or output. The call syntax of `select()` is

```c
int select(int nfds, bitmask readfds, bitmask writefds,
           bitmask exceptfds, timeval timeout)
```

`readfds`, `writefds`, and `exceptfds` are bitmasks of file descriptors through which the selecting process is interested in performing I/O. `nfds` indicates how many of the bits in the masks should be examined, and `timeout` is a representation of how long the process should block before timing out the select call if none of the file descriptors are ready for I/O. `select()` returns the number of descriptors ready for service, and returns bitmasks of the ready descriptors in place of the passed bitmasks.

The XINU kernel provides a different call for the same purpose, called `xselect()`. Its call syntax is detailed in section 3.2. The library code for `select()` marshals the data and passes it to the kernel routine, and performs the inverse operation when returning to the application code.

### 3.4 Abstract Devices

The device-dependent layer of the X server (ddx) maps from a set of abstract operations to device-specific calls for the implementation architecture. The device-independent layer (dix) uses the abstract operations provided by ddx to pass data between the server and its associated devices.
Because our goal is to make it possible for a single implementation to span diverse underlying systems, we approach the XINU port of these layers with the goal of building a set of architecture-independent functions. Experience shows that if the device abstractions provide a sufficiently well-designed, high-level interface, it is possible to devise a single “glue” module that connects the X server to the underlying operating system across multiple architectures. Keeping the glue module portable requires building better device abstractions into the kernel. The result is encouraging: this module allows a novice programmer to port the X server to a new architecture, once a full port of XINU is completed.

3.4.1 Keyboard

The X server interacts with the keyboard through events. An X keyboard event is a pair consisting of a key symbol (keysym) and an activity tag that tells whether the key was pressed or released. dix defines the set of keyyms used in X keyboard events. Many keyboards have modifier keys, such as SHIFT, CTRL, META, ALT, SUPER, and HYPER. The dix layer maintains a mapping function

\[ \text{SymTrans}(\text{keysym} \times \{\text{modifier set}\}) \mapsto \text{keysym} \]

On most keyboards, \( \text{XK}_d \) is the keysym for \( d \) and \( \text{XK}_D \) is the keysym for \( D \), so

\[ \text{SymTrans}(\text{XK}_d \times \{\text{SHIFT}\}) \mapsto \text{XK}_D. \]

This mapping function is obtained from the ddx layer at server initialization because each keyboard defines an instance of this function. For example, the shifted value of ‘.’ is ‘.’ on a Digital Equipment Corporation keyboard, while it is ‘>>’ on a Sun Microsystems keyboard. Their mapping functions then yield the values

\[ \text{SymTrans}_{\text{DEC}}(\text{XK}_\text{period} \times \{\text{SHIFT}\}) \mapsto \text{XK}_\text{period} \]
\[ \text{SymTrans}_{\text{Sun}}(\text{XK}_\text{period} \times \{\text{SHIFT}\}) \mapsto \text{XK}_\text{greater}. \]

Keyboards generate a hardware scan code for each key pressed, e.g. the Apple Macintosh generates a scan code of \( \frac{47}{16} \) for the ‘,’ key. It is the job of the ddx layer to translate from scan codes to keyyms. Since different keyboards can have different scan codes for the same key, each keyboard defines a function

\[ \text{CodeTrans}(\text{scancode}) \mapsto \text{keysym} \]

which must be implemented by the ddx layer for each distinct keyboard type.

We have defined a XINU abstract keyboard (skb) that provides a uniform keyboard interface across diverse hardware. The device driver for this keyboard returns keyyms as its abstract scan codes. The translation from hardware scan codes to keyyms is done by the device driver. Thus, \( \text{CodeTrans}_{\text{skb}}() \) becomes the identity function, and \( \text{SymTrans}_{\text{skb}}() \) is the same function regardless of the underlying keyboard hardware. The mapping function from scan codes to keyyms for a new keyboard is written only once, during the port of XINU. The primary benefit of this is that the keyboard device is hardware independent, and applications that use it are easy to port to different architectures.

Figure 11 shows the relationships between the various layers. The boxes are functional layers, and the arrows are labeled with the type of information passed between the layers. The translation performed at each layer is identified by the function within its box. \( I() \) is the identity function, and \( \text{AbsTrans}() \) is the function that maps from hardware scan codes to XINU abstract
scan codes. CodeTransSun() translates Sun Microsystems hardware scan codes to keysyms, and CodeTransXkb() performs a 1-to-1 mapping from abstract scan codes to keysyms.

The leftmost diagram represents a conventional keyboard interface for a Sun workstation under SunOS; the center diagram depicts the keyboard interface for the XINU abstract keyboard driver, and the rightmost diagram shows the result of the simplifying assumption that XINU abstract scan codes are equivalent to keysyms. The effect of this assumption is to exchange the computations done by the ddx and device driver layers with those in the Sun interface. This places the responsibility for implementing the CodeTrans() function inside the kernel. This eliminates duplicated effort on the part of programmers and reduces the likelihood of errors.

### 3.4.2 Mouse

XINU defines a set of hardware-independent mouse events. As the hardware mouse generates interrupts, XINU maps them into device-independent events and enqueues them for processing. The structure of a mouse event is shown in figure 12.

Although many hardware mice return information combining button status and movement, the definition of an event for the abstract mouse includes the requirement that only one item of interest is reported per event. If more than one button changes state, or if the mouse moves at the same time as a button changes state, the mouse driver generates and enqueues multiple mouse events.
struct ms_event {
    long ms_timestamp;
    char ms_type; /* BUTTON_DOWN, BUTTON_RELEASE, MOTION */
    char ms_button; /* XINU_LEFT, XINU_MIDDLE, XINU_RIGHT */
    char ms_d_x,
        ms_d_y; /* movement. positive advances to upper-right */
};

Figure 12: XINU mouse events

with identical timestamps.

### 3.4.3 Frame Buffer

The XINU abstract frame buffer is a special device in that it provides no functionality for reading or writing. Rather, it provides a set of control calls used for obtaining and setting information about the frame buffer characteristics.

The device driver for the frame buffer returns information about the height, width, depth, and total memory size of the frame buffer. In addition, it allows an application to map the frame buffer's memory into its virtual address space, to probe the color map, and to assign a color in a color map.

During the initialization phase, the X server opens the frame buffer and queries the device for its characteristic values such as the number of colors available and the dimensions of the screen. It then sets its runtime variables for the screen parameters, and stores pointers to the correct color or monochrome screen-drawing functions. In this way, a single copy of the X server executable can be run on machines with different screen sizes, and on color and monochrome machines.

### 4 Conclusion

We have examined a prototype implementation of the X11R4 sample server for the XINU operating system. As part of this examination, we have noted the system services the server requires, and how these requirements are met in XINU.

Our experience with the prototype leads us to conclude that it is feasible to support the server with a minimal kernel while providing an extended operating system interface with library software running outside the kernel.

We have also seen that abstracting the characteristics of a class of devices into a high-level event-oriented driver allows us to write applications that can make efficient use of a broad range of underlying devices. Such an approach allows a programmer to port applications rapidly, while still allowing for customization and optimization of the drivers for special devices or applications. As an example, after implementing and testing the abstract device drivers, we moved the X11R4 sample server from a Motorola 68020 platform to a MIPS 3100 platform in one day. We plan to explore this idea more extensively, and hope that the porting time for other applications that use the abstract devices can be similarly reduced.
References


