KERNEL DEVICE MANAGEMENT USING A HIGH LEVEL I/O PROTOCOL

by

FREDERICK DIXON SAMPLE

B.Sc., The University of British Columbia, 1980

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

THE DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

April 1984

Abstract

This thesis examines the advantages and disadvantages of using a high level I/O protocol for device management. It covers the implementation of this form of device management as an addition to the Verex operating system, the problems encountered, and their solutions.

Table Of Contents

List of Figures

Acknowledgements

I would like to thank the many people who contributed in one way or another to this thesis: Dr. David Cheriton, who proposed the original idea and supervised much of the implementation of the work; the other graduate students working with Verex, who gave advice and ideas; Gerald Neufeld, who supervised the writing and was very patient; and my family and friends, who provided encouragement.

Chapter 1
Introduction

The subject of this thesis is the investigation of the advantages and disadvantages of managing device communication using a high level data communication protocol, within the kernel of a message based operating system. Of particular concern are the effects on operating system portability, flexibility, and efficiency. The research work involved in this thesis includes the design and implementation of device handling routines using this model.

This research was done in conjunction with related research into data communication protocols at the University of British Columbia Department of Computer Science. A protocol of sufficient power and flexibility to handle a wide variety of input and output tasks was developed, and a subroutine package was written to provide convenient access to input and output services that conformed to the protocol [10].

1.1 Objectives

The aim of this thesis is to examine the possibility of using a high level I/O interface for communicating with peripheral devices. The expected advantages of this approach to device handling are dynamic reconfigurability, more efficient communication with devices, and the confinement of privileged I/O instructions to the kernel. Also, more of the machine dependancy in the system is confined to the kernel. A possible problem is the increase in the amount of code that is contained in the kernel. Determining the severity of this problem is one of the objectives of this research.

Another objective of this thesis is to describe the implementation of this form of device management as an addition to Verex [1], and to report on the problems encountered and their solutions.

1.2 Motivation

One of the prime motivations for using an I/O protocol is the improvement of device management. Current schemes are unstructured and often inefficient. The inefficiency of these schemes can severely limit real time applications due to the inability to guarantee that interrupts will be serviced within a certain time interval.

Implementing the protocol at the kernel level rather than having a device handler process enables direct communication between user processes and devices, thus reducing the message passing overhead. The I/O protocol message to the device, as it is intercepted in the kernel, is received immediately, eliminating the process switching overhead that would otherwise be required to send a message and receive a reply.

Verex device handling, largely inherited from Thoth [4], involves awakening a process to handle most interrupts. This is costly, so costly in fact that the maximum rate that a terminal can send characters to the cpu without any characters being missed is only 1200 baud. Many modern intelligent terminals can transmit programmed strings or screen readouts to the cpu, usually at the same rate as characters are received from the cpu. To take advantage of this facility on Verex, currently the terminals must be set to a rate of 1200 baud, a severe limitation on their performance. The Verex I/O protocol reads and writes in blocks, enabling a kernel implementation of this protocol to buffer characters. Hence a kernel level implementation of this protocol need not awaken a process for each interrupt. As the blocks can vary in size, the block I/O allows single character I/O as well.

Verifiability is one of the objectives of Verex. On many machines, I/O operations are privileged. Limiting the code which must run in privileged mode to the kernel improves the verifiability of the operating system, as the amount of damage that can be caused by a malfunctioning process is much less if it is not running in privileged mode. In particular, devices that access memory directly can overwrite kernel code, so the kernel must supervise all such devices to ensure its integrity. The original work done on verifiability with Verex aimed to exclude I/O operations from the kernel, so that the kernel would not have to be re-verified each time a device handler was added to the system. Supervision of direct memory access transfers requires knowledge of device function, hence excluding device dependancy from the kernel is impossible. Confining I/O operations to kernel code is also the major step in isolating the kernel from the rest of the operating system, which currently runs in the same address space as the kernel.

Above the kernel level, the kernel device management system appears as a server process. This is in accord with theoretical models which regard devices as external processes [6]. The kernel device management system extends Verex interprocess communication to these external processes.

Chapter 2
Research Background

Although device management software forms an important part of all operating systems, little has been written about its design. Perhaps this is due to the hardware dependant nature of device software. Brinch Hansen [6] referred to devices as "hardware processes", yet little work has been done in extending interprocess communication to devices.

2.1 Interprocess Communication

Interprocess communication has increased steadily in sophistication with the development of modern operating systems. Early operating systems provided very little in the way of communication between processes, as all but the lowest level system code was restricted to one process per program. As the idea of multiple processes per program became more popular, the methods of interprocess communication increased in power to meet the demand.

Interprocess communication serves two main purposes, data transmission and synchronization. The simplest form of data transmission is through the use of shared memory. Use of shared memory must be coupled with a method of synchronization, to avoid critical section problems. Semaphores [11], are the most basic form of synchronization. Message passing provides both synchronization and data transmission.

Shared memory requires the processes sharing memory have overlapping address spaces, which often incurs either protection difficulties or high computational overheads. The simplest method is to have the processes run in the same address space, but this limits all communicating processes to one address space, and leaves each process dependant on the correct behavior of all other processes. Less dangerous methods either incur large computational overheads or require special hardware [4].

The use of message passing for interprocess communication has been growing as new operating systems are developed. Message passing is a set of primitives for the exchange of packets of data, called messages, between processes. Message passing schemes can be divided into two categories: synchronous and asynchronous. Asynchronous message passing allows a process to proceed with execution after sending a message, whereas synchronous message passing requires the sender to wait until the receiver of the message has received it and sent a reply. The advantage of synchronous message passing is its simpler implementation, as message buffers need not be dynamically allocated in the kernel, resulting in greater reliability, since there is no danger of running out of message buffers [4]. Its disadvantage is the reduction of concurrency. However, concurrency can be achieved by the creation of more processes.

Verex message passing uses small, fixed length messages which are passed synchronously. The message passing primitives consist of a routine to send a message, a routine to receive a message, and a routine to reply to a message. A process sending a message blocks until that message has been replied to, or the process sent to no longer exists. A process may check to see if a message has arrived for it, or it may block until a message arrives. A process may await a message from a particular process, or from any process. The reply primitive passes a message to a process that has sent to the replying process, and unblocks the sending process.

2.2 Input/Output Protocols

Input/Output protocols are the subject of much research currently with the great expansion of the use of computers for communication. Standard protocols increase the portability and flexibility of the programs using them, as their range of possible application is increased.

Madnick [12] details the advantages of having a uniform presentation of a file system. His arguments are easily extended to providing a uniform interface to all types of input and output. Most modern operating systems provide a uniform file/device abstraction at the top level.

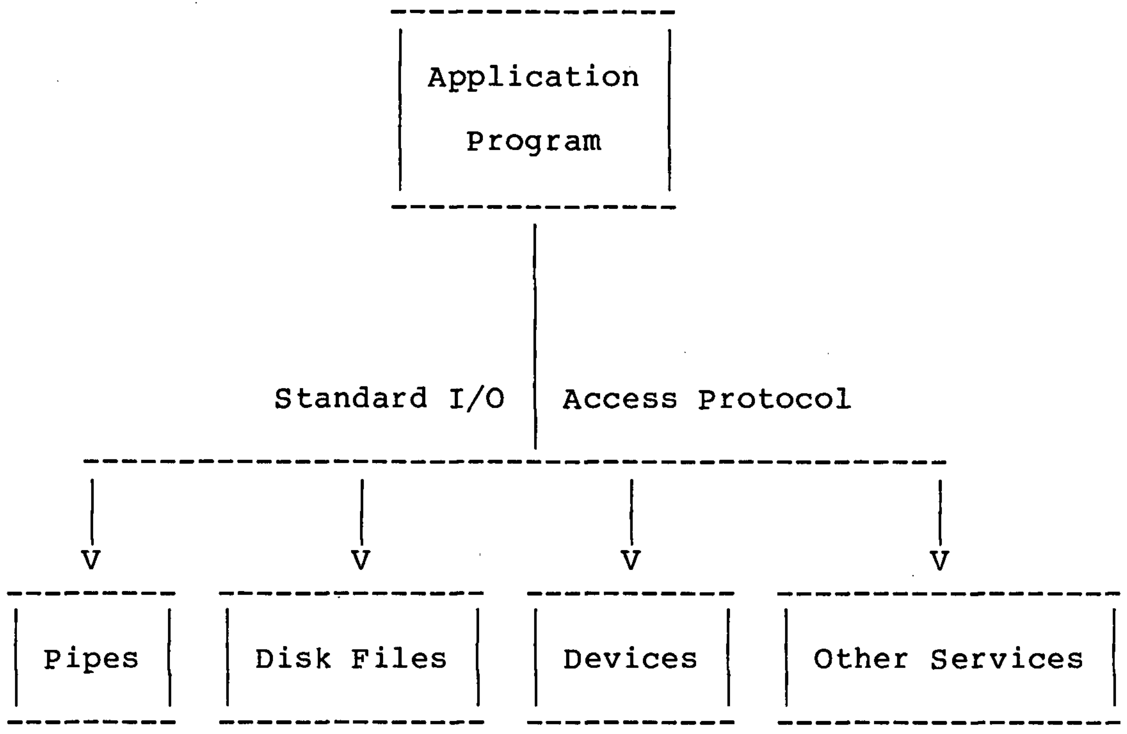

Figure 2.1 -- Uniform Access Protocol

Cheriton [10] has developed a universal file access protocol using message passing. This protocol is based on the client/server model, where a client process makes I/O requests to a server process which manages the particular file abstraction. As all servers communicate using the same protocol, programs using the protocol can be used with a wide variety of file/device abstractions.

2.3 Verex Device Management

The device management philosophy in the original Verex is to keep the code in the kernel to a minimum, and have processes handle the majority of the work. Hence the device management interface to the kernel is low level.

The kernel provides one device interface function, **.Await_interrupt**. A process uses this kernel operation to await an interrupt at a particular interrupt level. When an interrupt occurs at that level, the interrupt routine for that interrupt level is invoked, some calculations are performed, and the process awaiting the interrupt is unblocked. A process is not necessarily unblocked each time an interrupt is received. In this manner, interrupts that request simple action can be handled by the interrupt routine. After awaiting an interrupt and being unblocked, the handler process then deals directly with the hardware.

Response to frequent interrupts is hampered by the limited rate at which processes can be unblocked and run. This problem is avoided by putting code into the interrupt routines to buffer incoming data, and adding kernel operations to read the data. The problem with this method is that each fast device requires its own device input/output kernel operations.

2.4 Unix Device Management

Unix divides devices into two classes, **block** and **character**. Block devices must be random access, and must use 512 byte blocks. Character devices are devices that do not fit into the "block" class. Devices are identified by a sixteen bit number, which identifies the device driver, and the particular device handled by that driver [5].

Device driver software consists of the device's interrupt routine, and interface procedures. Device driver routines are accessed by using the upper eight bits of the device identifier to index a table of device functions. Character and block type devices use separate tables. Every device has an open and a close function. Character devices each have in addition a read routine, a write routine, and a special function routine. Block devices have a "strategy" routine which perform read, write, and control functions in a manner appropriate to the device.

Communication between the interrupt-driven part and the procedure called part of the device driver uses common variables, flags, and queues. Synchronization is accomplished through the use of interrupt priority levels, and the use of **sleep** and **wakeup** routines which allow the procedure to cause the calling process to block until it is unblocked by the interrupt routine.

Unix maintains a pool of buffers and supplies software for their management. These buffers are used by the block type devices to keep copies of frequently used blocks in memory. All block type devices have chains of buffers that are managed by the buffer management routines. Many routines are provided to implement the different ways in which buffered input and output can be done. Each buffer contains information on its status and the information it contains.

Unix also contains some provisions for transfers directly from devices into a users address space with non-standard block sizes. These devices are set up to be character devices which have their own buffers.

A user program calls standard functions to perform I/O. These functions access the system file table, which, among other information, contains the offset into the file at which the next I/O will take place, and a pointer to the file's **inode.** The inode contains the device identifier, which is then used to access the appropriate device driver routine.

Unix device management provides many of the features desired in device management systems, but is limited by several problems. The device access protocol is fairly uniform across all devices; all devices appear as files to the users, even to the point of having a particular location within the file system structure. The file access protocol is quite limited, however, which means that some control functions are difficult to make available at the application level. The lack of suitable interprocess communication forces device handlers to be within the kernel of the operating system, which causes the kernel to grow quite large. The device handling structure within the kernel is quite restrictive, which can cause difficulties in writing device drivers for devices which do not match Unix's idea of how devices should behave.

Chapter 3
Design

3.1 Design Criteria

The aim of this research is to examine the use of high-level protocols for device management, hence the primary design objective is to create a device management system which communicates with processes using a high-level I/O protocol. Devices should appear to be files implemented by a standard I/O server, so that a device can be used interchangably with another type of "file".

Another design aim is to provide a structured method of implementing a device driver, so that adding a new device is a simpler process. The device implementor should not be required to be an expert on the operating system in general. This enhances the portability of the operating system, as porting to a new machine requires new device drivers to be written.

The device interfaces should lose little of the flexibility provided by low-level interfaces. The implementation of a device driver for an unusual device should not require the implementor to write complicated code to outwit, or "program around", the device management system. This requires that the I/O protocol chosen be sufficiently flexible to accomodate widely varying types of input, output, and control information. For example, in the original Verex operating system, the simple primitives for device management were inadequate to control network interfaces, forcing the implementors to add kernel operations to communicate with them.

Real time response and efficiency should not be lost in achieving the above aims. An inefficient device management system is not likely to be used, especially if it cannot give adequate response to the interrupts of the devices it manages. For example, device managment schemes which involve scheduling a process to service an interrupt, a time consuming operation, cannot service high frequency interrupts.

Many operating systems require either regeneration or recompilation when the hardware configuration of the system is changed. Hardware configuration, as it is used here, defines the type of devices and interfaces connected to the computer, and their arrangement in I/O address space. This device management system will attempt to eliminate the necessity of recompilation or regeneration. Changing the hardware configuration will only require editing a setup file, as long as no device for which device drivers do not exist are added. This contributes to the reliability of the system, as compilers change with time, and there is no guarantee that a recompiled version of a system will behave correctly [2]. Modification of a setup file is also a much simpler, faster, process than recompilation. This method of reconfiguration also saves disc space in some circumstances, as operating systems which require recompilation to change hardware configuration require that a separate core image of the system be kept on file for each different hardware configuration in which the system may be run. An example of this is Data General's MRDOS operating system, where the number and type of each device is specified to a system configuration program, which generates a core image which will only run with the given hardware configuration.

Some classes of devices have a number of different modes in which they can operate, which require different interface and interrupt routines. An example of this is some terminal/network interfaces which can operate synchronously or asynchronously. It is advantageous not to need to reboot when a change of mode is desired. This "dynamic reconfigurability" is closely related to the "regeneration-free" reconfiguration described above.

3.2 Environment

The device management system described here is designed to work with kernel based multi-process operating systems with message based interprocess communication. The reason for this restriction is that a major element in the design is the extension of interprocess communication to devices. Systems with primitive interprocess communication would gain little in having it extended to devices. The system is particularly aimed at those systems that use a message based

I/O protocol of the client and server type, as they have subroutines designed to take advantage of high level communication with devices. Some aspects may be transferrable to other types of operating systems.

3.3 Design

The design of the device management system is broken into three parts: the services provided by the system, the interface between processes and the system, and the internal function of the system.

3.3.1 Services

The protocol used defines, to a certain extent, the nature of the services provided by the device management system. Most protocols, however, allow a fair degree of flexiblity in the exact operations performed by requests. The desired behavior of the system has been defined above in the design criteria. This section will detail the functionality required to achieve the desired behavior.

3.3.1.1 Device Creation

In order to achieve a kernel that is independant of the hardware configuration of its host computer, all hardware configuration dependant information must be removed from the kernel. Devices, therefore must be initialized and activated by requests from processes. In general, these device creation requests will come from the system processes that use the particular devices, for example, the file server will issue the device creation requests for its disk drives and other devices.

As no hardware configuration dependant information is contained in the kernel, the relevant information must be contained in the device creation request. This information will consist of such things as interrupt vector addresses, device control register addresses, and other information about devices which change with hardware configuration. The exact nature of the information will depend on the hardware I/O structure of the host computer.

Device initialization and activation is a device dependant operation. To allow the necessary flexibility to implement a wide range of device types, each type of device should have its own device creation routine. Selection of the appropriate routine when a device is created must depend on an indication of device type in the creation request. A simple protocol is required to map device types onto identifiers, which can be implemented as a set of constants shared by the source code of the device management system and the source code of the system processes using the devices.

3.3.1.2 Device Removal

Dynamic reconfigurability requires a method of releasing devices so that their mode of operation can be changed. When a device is released, the device management system releases its data structures and reserved hardware (see Protection, below), and terminates pending I/O on the released device. The device removal will not always be prompted by a request from a process, as a process may be destroyed or abort, in which case any devices it owns should be removed. This implies that the device management system must detect process destruction. Process destruction detection can take the form of periodic checking for existance, or notification upon destruction. Notification upon destruction is more difficult to implement, but is more reliable as the device is removed as the process is destroyed, not sometime later. Because of its greater reliablity, the notification method was chosen for implementation.

3.3.1.3 Protection

One of the basic assumptions when designing kernel software is that processes are inherently unreliable, and possibly malicious. For this reason, manipulation of devices must be subjected to checks and restrictions.

One area that must be protected is interrupt vector locations. If a device is created with the same interrupt vector location as a device that is already active, the new interrupt routine would end up servicing interrupts from both the old device and the new device. This would likely cause confusion, hence the vector locations must be checked at device creation time.

Some devices, such as disc drives, contain precious information, and should not be accessible to user processes, whereas other forms of devices, such as terminal interfaces, are less sensitive. When a creation request is received for a sensitive device, the type of the process sending the request should be checked. The request should be rejected if the process does not meet some security criterion, such as being a system process.

Similarly, some devices are insensitive to read and write requests, hence read and write requests can be allowed from any process. More sensitive devices, however, should check each read and/or write request to ensure that it comes from an acceptable process. The most useful acceptance criterion for sensitive devices is that the request comes from the same process who created the device.

3.3.2 Interface

3.3.2.1 Communication

A basic component of the device management system is a means of communication between processes and device software via messages. To maintain compatibility with I/O library routines, device communication must appear exactly the same as interprocess communication. This requires that the kernel primitives implementing message passing must detect messages that are intended for devices and pass them to the appropriate code.

The method used to determine if a message is intended for a device or a process cannot depend on the contents of the message, as message contents should be unrestricted. This leaves the process identifier of the process to which the message is sent as the only source of information to discern the proper destination of a message. Therefore one or more process identifiers must be reserved for device communication.

There are a number of ways in which process identifiers could be used to indicate that a message is intended for a device. The primary factor influencing the selection of a particular method is the amount of overhead added to the message passing, as message based operating systems generally have a high volume of message traffic [10].

The simplest possible method is to reserve one unique process identifier to indicate device communication. This has the advantage of being easy and efficient to detect, requiring only a single comparison. A disadvantage of this method is that it eliminates any possibility of using information in the process identifier for other purposes such as device class selection. This disadvantage is minimized by the fact that I/O protocols use fields in the message to identify a particular object. Two level systems, which select both on the process identifier and the object identifier in the message, are impossible with this scheme.

Another possible method is to reserve a group or range of identifiers. The particular identifier in the group can then be used to give some information about the device. This allows greater flexibility in device specification, but incurs a small performance penalty, and is more complicated.

The single reserved id method was chosen for implementation, because of efficiency and simplicity considerations. The two level selection possible using a range of reserved process identifiers gives a wider range of device selection, but requires more code and takes more time. The object identification field in the message seems to give sufficient device specification range.

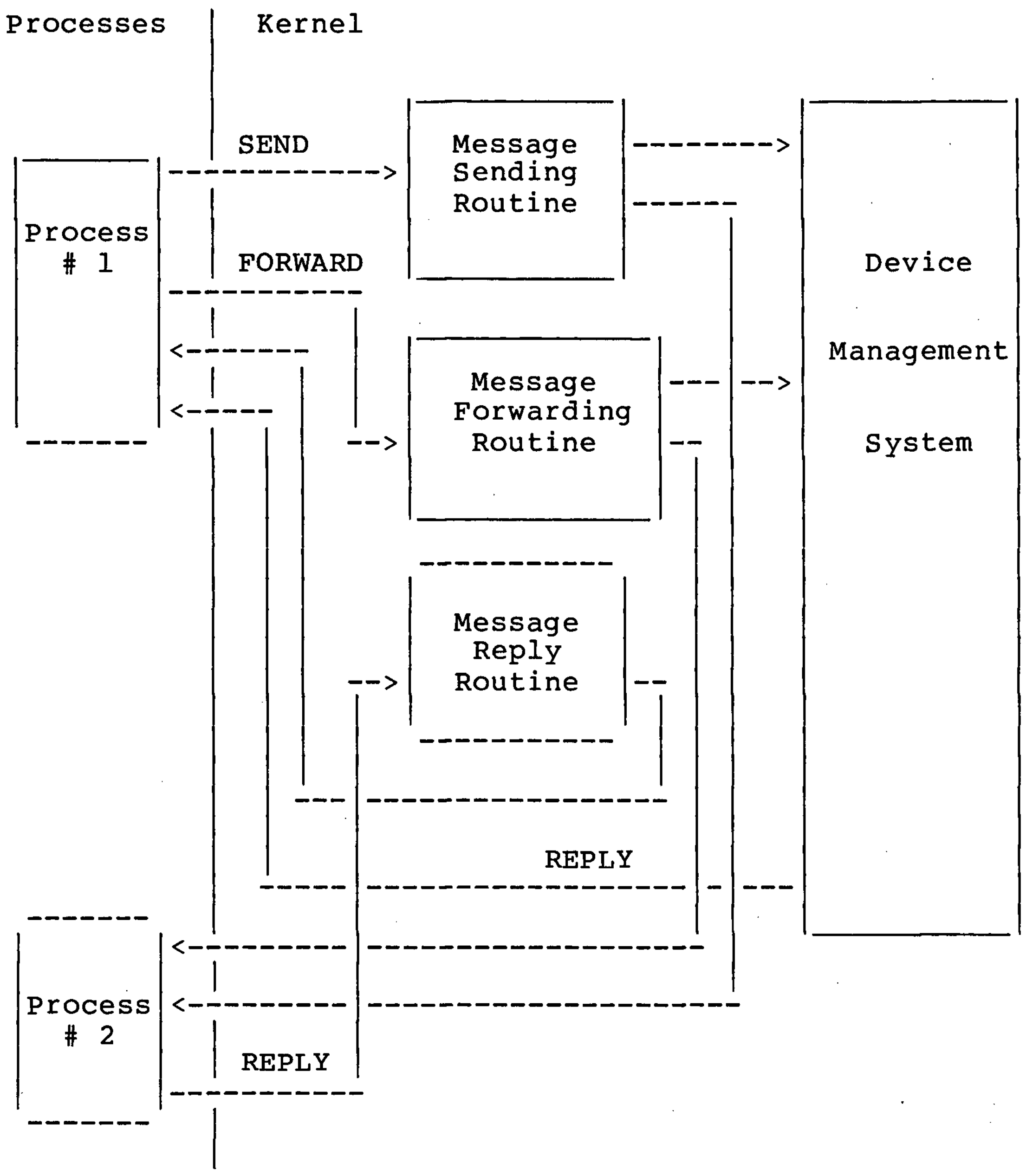

Figure 3.1 -- Process Device Communication

Once a message has been identified as being intended for a device, the appropriate device handling code must be selected. The selection of code depends on the type of device being addressed, and the service being requested.

3.3.2.2 I/O Protocol

The remainder of the interface is largely defined by the I/O protocol chosen. Extensions will probably be required to the protocol for the specification of the hardware information in the device creation request.

3.3.3 Internal Design

A device driver can be separated into two distinct parts: interrupt driven and request driven. Routines in the request driven section are executed in response to an I/O protocol request from a process, while interrupt driven routines are executed in response to an interrupt from a device. Communication between the two sections must be through the use of shared data structures, as message passing and process synchronization do not exist below the kernel level. For example, in response to a read request, the request driven section of a device driver might check a buffer for data, and block the requesting process if there was none. In reponse to an incoming data interrupt, the interrupt driven section checks the shared data structure to see if a process is blocked waiting for input, and unblocks it if there is, otherwise buffering the data.

3.3.3.1 Data Structures

In choosing the form of data structures used, one trades off flexibility against overhead. The more flexible memory management schemes have a high space overhead, and an even higher execution time overhead. At the kernel level, it is important to minimize overhead, so the more compilcated methods must be rejected. Static allocation has no overhead, but rules out dynamic reconfiguration. A suitable compromise between these two extremes is dynamic allocation of memory in pieces of a few fixed sizes.

A data structure is required to describe the device to the device management system, and to store device dependant information. This data structure is used to select the appropriate device dependant code to service I/O requests for the device. Access to this data structure should be fast, as it must be accessed for every device request. This suggests the use of a table of pointers to device state vectors, with the file identifier of the device used to select the element of the table. A simple indexing scheme is undesirable,as it allows possible confusion between a device recently removed, and a new one that has been created with the same index. This problem can be avoided by using the low order bits of the file identifier as an index, and the high order bits as a sequence number. The time required for the sequence number to recycle makes confusion unlikely.

Certain requests in I/O protocols can be serviced with little reference to device specific information. These requests do not require device specific routines, but can be handled by general routine with a little device specific information from the device state vector. Space must be allocated in the state vector for the needed information. The decision of which requests to implement in this fashion is based on optimization of memory requirements.

The device state vector must also contain space for device specific information storage. As the amount of needed storage varies from device to device, and the state vector size is fixed, deciding how much "extra" space to leave is a problem. The best solution is one that minimizes the memory space used, which can be determined experimentally. More than one state vector can be allocated for those devices which require more storage than is allowed in a single state vector.

3.3.3.2 Request Driven Routines

As mentioned above, some requests require little in the way of device specific information to be serviced. These mainly take the form of simple querys and parameter changes. Flexibility considerations require the use of device specific code to implement some I/O requests. These routines are accessed through the device state vector, which contains their addresses. The requests are likely to require device specific code are those which perform the more complicated functions, i.e. data communication and device control. To ensure flexibility it is better to err on the side of too many device specific routines. The overhead of allowing a routine to be device specific is only one address in the device state vector, and a small time overhead to access the descriptor and the address of the routine. Devices that have similar requirements can share the same request service code.

3.3.3.3 Interrupt Driven Routines

Very few restrictions should be placed on the way in which interrupt routines are written, as they must deal with widely varying kinds of hardware, in as efficient a manner as possible. The only structure imposed is that of communications, that the interrupt driven routines communicate with the request driven routines through the shared memory of the device state vectors and buffers.

3.3.3.4 Utility Routines

To make the device management system easy for the implementor, a wide range of utility routines should be provided. Use of utility routines also increases the reliability of the device drivers, as objects are handled in a standardized, tested fashion.

Memory management is one area in which utility routines are important. Routines should be provided for the allocation and freeing of device descriptors and buffers. Initialization of standard fields in the device descriptor is another useful utility. Routines should be provided to manage such "built in" data structures as interrupt vectors, minimizing the machine dependant data that the implementer must learn.

Many devices share common properties, which allow the use of common routines for controlling them. For example, most of the software used to map terminal interfaces onto the I/O protocol is common to all terminals interfaces, regardless of the hardware protocols used. The common parts should be provided as utility routines and documented, to minimize the work of implementing a driver for a new type of terminal interface.

Another class of utility routines that should be provided is no-operation (NOP) routines. These routines are used to fill slots for device dependant request service routines, when the particular service is impossible or meaningless for a device. For example, performing a read operation on a printer interface is, in general, meaningless. In this case, the slot for read request service would be filled with a pointer to a routine that merely returns a reply that indicates that the device is not readable.

Processes that communicate with devices usually must be synchronized in some fashion with the interrupts from the devices. Routines must be provided to block a process and restart it after an event. These routines usually exist in some form or other in any multiprocessing kernel, and often may be used without modification when a device management subsystem is added.

Chapter 4

Environment

4.1 Verex Process Management & Communication

Verex is a message based operating system. Operations for process creation, and interprocess communication are provided by the Verex kernel. The kernel also provides primitive process scheduling.

The two operations used for process creation are:

```
new_process_id = .Create_process()
```

which allocates a process descriptor and returns its id; and

```
.Init_process( id, stack_size, ... )
```

which initializes the process descriptor and schedules it to be run.

Interprocess communication uses small, fixed size message buffers, which reside inside the kernel. The kernel provides operations to read and write the message buffers from the process' address space. The basic communication operations provided are:

```
receiver_id = .Send_msg( id )
```

which sends a message to a process whose id is given as a parameter, and awaits a reply;

```
sender_id = .Receive_msg( [id] )
```

which receives a message from the specified process, or any process if no id is given, if a message has been sent the id of the sending process is returned, otherwise an indication that no message has been sent is returned;

```
.Await_msg( id )
```

which waits for a message to be sent to the invoking process from the process specified, or any process if none is specified, and then returns the id of the sending process;

```
.Reply_msg( id )
```

which replies to the sender specified; and

```
.Forward_msg( from_id, to_id )
```

which passes a message to the process specified by the parameter "to_id", as if it was sent by the process specified by the parameter "from_id".

Message passing is synchronous, as a sender must wait until his message has been received and replied to before proceeding. The only asynchronous event possible is process destruction:

```
.Destroy( id )
```

which destroys the process specified by id, if it has the same user as the invoking process, or the user of the invoking process is the system user.

The kernel process scheduling maintains multiple prioritized queues of processes, and ensures that the highest priority ready process is always running. A process being added to a queue of a given priority level is added at the end, creating round-robin scheduling for process at the same level. Higher level scheduling is done external to the kernel, through modification of process priority.

At the kernel level, the routines .Block, .Unblock, and .Block_and_unblock are used to remove and add processes to the ready queues. These routines are accessible to the device management software to schedule processes waiting for I/O.

4.2 The Verex I/O Protocol

The Verex I/O protocol is designed to provide standard interaction with a wide variety of data storage and communication facilities. The protocol is object based and connectionless, using the client-server communication model.

All services using the protocol are mapped onto a standard view of a "file". A standard file is made up of "blocks" which may vary in length. Each block has associated with it a number that defines its logical position. The way in which the data is divided into blocks is not defined by the I/O protocol, and can be selected to match the characteristics of the service.

Operations provided by the I/O protocol are executed on "instances" of files. A file instance is a currently active entity, identified by the identity of its server process, and a file instance identifier provided by the server. The main distinction between files and file instances is that instances are dynamic, existing only for the duration of some activity, whereas files may exist through many instantiations.

To allow the use of widely differing I/O facilities, the protocol defines standard descriptions of file attributes. This allows the client to treat a file instance in a manner appropriate to its type. These attributes define the limitations on the operations that can be executed on the file. For instance, the **WRITEABLE** attribute defines whether a file can be written to, and can be used to implement write protection on files.

The message as provided by the interprocess communication primitives is the basic unit of the protocol. The type of request is specified by the **.REQ_CODE** field of the message. This semantics of the rest of the message is defined according to the type of request. The request codes defined by the I/O protocol are: **.CREATE_INSTANCE**, **.RELEASE_INSTANCE**, **.QUERY_INSTANCE**, **.READ_INSTANCE**, **.WRITE_INSTANCE**, and **.SET_INSTANCE_OWNER**. The reply message from these requests has a standard **.REPLY_CODE** completed, or its reason for failure.

The **.CREATE_INSTANCE** request creates an instance of a file and returns an instance identifier and the file's attributes. The nature of the instance created depends on server dependant information used to specify the file desired, and a standard "usage mode" which defines the manner in which the file is to be used.

The .RELEASE_INSTANCE request informs the server that a client is finished with a file instance. The server then performs whatever cleanup operation is associated with the release of the file instance. A field in the release request specifies the "release mode" of the instance. Use of this information is server dependant, and is usually used to indicate whether or not permanent data should be updated.

The .QUERY_INSTANCE request returns the same information as the .CREATE_INSTANCE request, without creating a new instance. The file instance queried is specified by the files instance identifier.

The .READ_INSTANCE and .WRITE_INSTANCE requests are used for data transfer between the client and the file instance. The block number, the number of bytes to be transferred and the location of the data are specified by fields the request message.

The .SET_INSTANCE_OWNER request changes the owner of the specified file instance to a new owner specified by a field in the request message. This allows instance ownership to be transferred from one process to another. This is necessary as some servers allow certain operations on an instance to be executed only by the instance's owner.

Two additional requests used by some servers are the **.CONTROL** and **.QUERY** requests, which are used to request operations and server information not provided for by the standard I/O pronotcol. For example, the file server uses these requests to perform such operations as track formatting, and the terminal server to return information about terminal characteristics. For a more complete description, the reader is referred to Cheriton [10].

4.3 Verex I/O Servers

Through the use of a standardized protocols, the Verex operating system has evolved to the point where all operating system services, aside from the basic few provided by the kernel, are provided by server processes. This has resulted in a highly modular system, which is easily reconfigured to suit its purpose, as the selection of servers invoked when the system is started can be varied simply by modification of a setup file.

The use of the I/O protocol has led to the creation of many I/O servers which use it to interact in a standard fashion. As all servers use the I/O protocol, the type of file being used is transparent to the application, aside from special control functions possible on some devices. The protocol has been successfully applied to communication with file servers, local and long haul networks, terminals, inter-user mail, and pipes.

A name server is used to relate the symbolic name of a service to the process id of the server process. This provides great flexibility in the redirection of the output from programs. For example, the output from the editor can be sent directly to the mail server, allowing the convenient composition of mail messages with a visual editor, without the use of an intermediate storage file.

Chapter 5
Implementation

This chapter will discuss the implementation of the ideas discussed in the previous chapters, as a part of the Verex operating system running on the Texas Instruments TI990/10. Device dependant code will be discussed in the appendices. A guideline for device implementation will be found Appendix A and an example device driver in Appendix B.

In order to implement kernel devices, changes must be made to the original kernel code for process creation and deletion, and message passing. Process creation must be changed to insure that the special **DEVICE_SERVER** id is not given to a process. The process destruction routine is altered so that a device being written or read by a process to be destroyed is returned to an idle state. The message passing routines must be altered to route messages sent or forwarded to the **DEVICE_SERVER** to the device server routine. Also, code must be added to the system initialization to set up kernel device data structures.

The basic kernel device data structures are: a table of device state descriptors; a buffer pool; and a table of the device creation functions for different device classes. The necessary routines for handling these data structures are: device descriptor allocation/release routines; a routine for mapping from device ids to descriptor addresses; buffer pool management routines; and a routine to call the device creation routine appropriate given its device class.

The device descriptor contains device independant information as well as space for device dependant information. The device independant routine includes: pointers to the code to be executed for read, write, control, and release requests; pointers to code to restart the device after power failure; information about the device, such as its block size; and information about the processes using the device.

The device descriptor allocation/release routines were kept simple to minimize code. As device creation and deletion are rare events, the routines need not be particularly fast. The allocation algorithm used is a simple sequential search through the device descriptor table for an unused descriptor. If no unused descriptor is found, an attempt is made to clean up resources allocated to devices whose owner processes no longer exist.

The central routine to the device server code is Send_device, which interprets the kernel device request in a manner similar to a standard server. The major difference is that the sending process is not replied to, but is either blocked or returned to depending on the device and the request. If the request is forwarded, the sending routine is unblocked unless the reply code is .NO_REPLY, and the forwarder is always returned to, whereas if the request is not forwarded, the sender is returned to unless the reply code is .NO_REPLY, in which case it is blocked.

Send_device calls the routine appropriate to the .REQUEST_CODE of the message. Device requests conform to the Verex I/O protocol, and are identical to standard server requests, except in the case of the .CREATE_INSTANCE request, where additional information must be specified. The information required are: the service type, which specifies the particular hardware device and how it is to be used; and hardware location information, which in the case of the TI/990 consists of the interrupt level and CRU address. The service type is used as an index into a table of functions to select the appropriate device creation/initialization routine.

Chapter 6
Results

The implementation of the device managment system as a part of the Verex operating system has proceeded to the point where it can be used as part of a production system. A version of the system incorporating the device management system has been in everyday use for four months, and has proved to be very reliable.

Not all devices have had their device drivers modified to work with the new system, so current systems use a combination where some devices are handled using the new system, and others still use the old. Devices are being converted to the new system as programmer time permits. The system commonly in use uses the device management system to drive all terminal and printer interfaces, leaving only the disc controller interface and the x.25 network interface unconverted.

Response to interrupts has been, as predicted, greatly improved. The terminals interfaces can now send characters to the cpu at 19200 baud, while dropping only the occasional character. The reason for the characters dropping is not due to the device management system, but to the fact that Verex disables all interrupts when executing kernel code. A possible future project is to modify Verex so that kernel code can run with interrupts enabled. No characters are missed at 9600 baud. The maximum rate of output to terminals has decreased by about ten percent. This is due to the small output buffer size. This can be cured by enlarging the output buffer size, and will be done when time permits.

As the terminal interfaces are now managed by the device management system, the high-level terminal server has been removed from the system address space. This has resulted in a twenty percent decrease in the size of the system, which is a great advantage as the address space limitation of the machine was beginning to make system expansion difficult. Removal of the terminal server from the system address space has allowed it much more room for expansion.

The high level interface provided by the device server has facilitated the addition of new features to the terminal server, through the use of control requests.

Chapter 7
Conclusions

The implementation of the device management system as a part of the Verex operating system has demonstrated the feasability and merit of communicating with devices using a high-level I/O protocol. Device drivers were written for a number of different device types, proving the flexibility of the method.

The extensibility of the ideas has yet to be proven, as Verex is the only operating system in which they have been implemented. It is clear that this form of device management could be applied to similar operating systems such as Thoth, but whether operating systems with primitive interprocess communication can benefit from it remains an open question.

As is usually the case in implementation, unforseen problems were encountered, requiring modification of some of the original ideas. It is likely that devices exist for which a device driver would be difficult to implement with this system. Fortunately, the implementation is easy to modify.

Appendix 1

Kernel Device Implementation Guideline

This appendix gives a step by step guideline for adding a device driver to the Verex operating system kernel. Familiarity with the Verex I/O protocol and knowledge of how to compile and boot a new system is required of the implementor. Other knowledge about the system and the kernel should be unnecessary.

A device driver in this system consists logically of two parts; the interrupt driven routines and the request driven routines, which communicate through the shared device state descriptor(s) and optionally through shared buffers.

Request Service Routines

When a request is sent to the device server pseudo-process, the appropriate routine is called to service that request. All request service routines are subroutines written in the Zed language, with parameters as described below. Some requests, such as **.QUERY_INSTANCE** are device independant and are handled by device independant code. The device dependant request service routines are identified in the device's descriptor.

The exception to this pattern is the routine which responds to the **.CREATE_INSTANCE** request, which occurs before the device descriptor is allocated. This routine uses the **SERVICE_TYPE** field of the creation request as an index into a static array of device creation routines. When a new device is added, the assembler module Create_function, is modified and re-assembled. This module specifies the array of device creation functions, and an external variable, Max_service_type, which indicates the length of the array. The name of the device creation function of the new device is added to the end of the array, and Max_service_type is increased by one. Note that a code for a device class can be excluded from the system by commenting out its entry in the Create_function table and replacing it with a zero word. This will cause the routine Create_device to return a reply code of **.ILLEGAL_REQ** to a device creation request for that device class.

The device creation routine is called with two parameters: a pointer to the requesting message, and a pointer to the process descriptor (PD) of the requesting process. Its function is to allocate and initialize all data structures used by the device. This will include at least one device descriptor, and possibly buffers provided by the buffer support routines. The device descriptor is allocated by the routine Alloc_device, and freed by the routine Free_device. The values of the fields in the descriptor are initially undefined and must be set by the device creation routine. Buffers are allocated by the routine Alloc_buffer, and freed by the routine Free_buffer. The buffers have a fixed size of 80 bytes.

A device descriptor is defined by the Zed template:

```
template DEVICE_STATE
  {
    word         REGISTER0;        \ R0 in interrupt registers
    word         REGISTER1,        \ R1
                 REGISTER2,        \ R2
                 REGISTER3,        \ R3
                 REGISTER4,        \ R4
                 REGISTER5,        \ R5
                 REGISTER6,        \ R6
                 REGISTER7,        \ R7
                 REGISTER8,        \ R8
                 REGISTER9,        \ R9
                 REGISTER10,       \ R10
                 REGISTER11,       \ R11
                 REGISTER12;       \ R12
    word         RETURN_WP[];      \ R13 (reserved)
    unsigned     RETURN_PC[](),    \ R14 (reserved)
                 RETURN_ST;        \ R15 (reserved)

    unsigned     READ_FUNCTION[](),
                 WRITE_FUNCTION[](),
                 CONTROL_FUNCTION[](),
                 RESTART_FUNCTION[](),
                 RELEASE_FUNCTION[](),
                 READ_CLEAR_FUNCTION[](),
                 WRITE_CLEAR_FUNCTION[](),
                 FILE_TYPE,        \ for respones to
                 IN_BLOCK_SIZE,    \ .QUERY_INSTANCE requests
                 OUT_BLOCK_SIZE,
                 WRITER_ID,
                 READER_ID,
                 USED,
                 DEVICE_OWNER,
                 FILLER:[2];

  }
```

The "FUNCTION" fields are initialized to point to the appropriate code to service the request. The fields **FILE_TYPE**, **IN_BLOCK_SIZE** and **OUT_BLOCK_SIZE** must be initialized in accordance with the Verex I/O protocol specifications for the **.QUERY_INSTANCE** request to work properly for the device. The field **USED** is used by the device descriptor management routines and must not be altered. The fields **READER_ID** and **WRITER_ID** may be used for keeping track of the ids of processes reading and writing the device. The field **FILLER** is three words of uncommitted space which may be used as the implementor desires.

The device creation routine must also initialize the interrupt vector, and initialize the device hardware. The interrupt vector is initialized with the routine **Add_device** which takes the device descriptor, the interrupt routine, and the interrupt level as parameters.

The device dependant request service routines handle the **.READ_INSTANCE**, **.WRITE_INSTANCE**, **.CONTROL**, and **.RELEASE_INSTANCE** requests. The appropriate fields in the device descriptor are initialized to point to these routines at device creation time. The routines servicing these requests are called with three parameters: a pointer to the request message, a pointer to the device descriptor, and a pointer to the PD of the requesting process.

Also needed are a **RESTART_FUNCTION**, a **READ_CLEAR_FUNCTION**, and a **WRITE_CLEAR_FUNCTION**. The restart function is called when the device is created, to initialize its hardware, and after power failure to restart the device. The read/write clear functions are used to notify the device driver that a process reading or writing it has been destroyed. These routines are called with the device descriptor as a parameter. Standard routines for some of these functions are provided as utility routines at the kernel level.

The path of communication with the device is through its read, write, and control functions. Data transfer to and from the device usually occurs after a device interrupts. The data can be stored in a buffer while awaiting the interrupt. The exact data transfer method is left up to the implementor. The control function can be used for special hardware functions, such as disk track formatting. Control functions are device specific, hence their definition is left to the implementor.

Utility Routines

Here is a detailed description of the utility routines available:

Add_device

Definition:

```
   Add_device( interrupt_level, routine[](), word device[] )
```

Description:

Add_device initializes the interrupt vector so that the routine pointed to by **routine** will be called with the workspace pointer pointing to device when an interrupt a the specified level occurrs. The parameter interrupt_level the machine handles.

Alloc_device

Definition:

```
   unsigned Alloc_device()
```

Description:

Alloc_device returns the number of a free device descriptor, which is used as an index into the external table Device_table. If no device descriptors are available, .MAX_UNSIGNED is returned.

Alloc_buffer

Definition:
```
   word Alloc_buffer()[]
```

Description:

Alloc_buffer returns a pointer to a free buffer, 80 bytes in length. If there are no free buffers, .NULL is returned.

Free_buffer

Definition:
```
   Free_buffer( word buffer[] )
```

Description:

Free_buffer frees the buffer passed to it.

Not_readable

Definition:
```
   unsigned Not_readable( word req[], word device[], word sender[] )
```

Description:

A dummy routine to return .NOT_READABLE to a .READ_INSTANCE request.

Not_writeable

Definition:
```
   unsigned Not_writeable( word req[], word device[], word sender[] )
```

Description:

A dummy routine to return .NOT_WRITEABLE to a .WRITE_INSTANCE request.

Illegal_req

Definition:
```
   unsigned Illegal_req( word req[], word device[], word sender[] )
```

Description:

A dummy routine to return .ILLEGAL_REQ to a .READ_INSTANCE request.

Interrupt Routines

This section is machine dependant. The method described here is for the TI990. The interrupt routine for a device is called whenever an interrupt is received from that device. On the TI, the code for an interrupt routine is called using a simulation of a "blwp" instruction using an offset into the interrupt vector. The workspace pointer that is loaded in the instruction points to the device descriptor for that device, hence the device state descriptor contains the register set for the interrupt routine. This enables fast, easy access to the device descriptor when an interrupt occurs. To access the part of the device descriptor that is not in the register set, the "stwp" instruction can be used to load the workspace pointer into a register, for use as a base address for indexing.

The primary utility routine used by interrupt routines is accessed through the external entry .Start_handler_entry. This routine is used to unblock a process, and is called with the branch instruction, i.e.:

```
    b .Start_handler_entry \ unblock waiting process
```

The calling routine should have a pointer to the PD of the process to unblock in register 9. To return from an interrupt when a process is not unblocked, use the "rtwp" instruction.

The use of the various registers is left up to the implementor. The routine .Start_handler_entry uses registers

7, 8, and 11, hence they are usable only for temporary storage. Registers 13, 14, and 15 are used to store the return address, workspace pointer, and status, and cannot be used for data.

Appendix 2

Example Device Driver

This appendix contains the source code for a simple EIA terminal interface. It is intended to be used as an example to clarify the device implementation guideline. An explanation of the function of each routine is included before the routines code.

```
template EIA_STATE
  {
    word          EIA_READER[];    \ R0 in interrupt registers
    unsigned      IN_BUF{},        \ R1
                  IN_BUF_END{},    \ R2
                  IN_PTR{},        \ R3
                  OUT_PTR{},       \ R4
                  OUT_BUF{},       \ R5
                  OUT_BUF_END{};   \ R6
    unsigned      REGISTER7,       \ R7
                  REGISTER8,       \ R8
                  REGISTER9;       \ R9
    word          EIA_WRITER[];    \ R10
    unsigned      REGISTER11;      \ R11
    unsigned      CRU_BASE;        \ R12
    word          RETURN_WP[];     \ R13
    unsigned      RETURN_PC[](),   \ R14
                  RETURN_ST;       \ R15

    unsigned      READ_FUNCTION[](),
                  WRITE_FUNCTION[](),
                  CONTROL_FUNCTION[](),
                  RESTART_FUNCTION[](),
                  RELEASE_FUNCTION[](),
                  READ_CLEAR_FUNCTION[](),
                  WRITE_CLEAR_FUNCTION[](),
                  FILE_TYPE,                  \ for respones to
                  IN_BLOCK_SIZE,              \ .QUERY_INSTANCE
                  OUT_BLOCK_SIZE,             \ requests
                  WRITER_ID,
                  READER_ID,
                  USED,
                  DEVICE_OWNER,
                  DROPPED,
                  SPURIOUS;
  }
```

This is the template which defines the device dependant fields in the EIA device descriptor.

```
Eia_create( word req[], word sender[] )
  {
    template    EIA_STATE, CREATE_DEVICE_REQUEST,
                .CREATE_INSTANCE_REPLY, .PD;
    extrn       Eia_read(), Eia_write(), Not_readable(), Eia_interrupt,

                Not_writeable(), Eia_restart(),
                Illegal_req(), Device_table[][],
                Mux_read_clear(), Mux_write_clear(), Mux_release();
    word        device[];
    unsigned    device_id;

    if( ( device_id = Alloc_device() ) == .MAX_UNSIGNED )
        return( .NO_MEMORY );

    device = Device_table[device_id];
    DEVICE_OWNER[device] = .ID[sender];
    READ_FUNCTION[device] = ( .FILE_MODE[req] & .READ ) ?
        &Eia_read : &Not_readable;
    WRITE_FUNCTION[device] = ( .FILE_MODE[req] & .CREATE ) ?
        &Eia_write : &Not_writeable;
    READ_CLEAR_FUNCTION[device] = &Mux_read_clear;
    WRITE_CLEAR_FUNCTION[device] = &Mux_write_clear;
    CONTROL_FUNCTION[device] = &Illegal_req;
    RELEASE_FUNCTION[device] = &Mux_release;
    RESTART_FUNCTION[device] = &Eia_restart;
    IN_PTR[device] = OUT_PTR[device] = IN_BUF[device] = Alloc_buffer();

    IN_BUF_END[device] =
        IN_BUF[device] + BUFFER_SIZE - pun( unsigned[], 1 );
    EIA_READER[device] = EIA_WRITER[device] = 0;
    CRU_BASE[device] = HARDWARE_LOCATION[req];

        Add_device( INTERRUPT_LEVEL[req], &Eia_interrupt, device );
    Eia_restart( device );

    .FILE_SERVER[req] = DEVICE_SERVER;
    .FILE_ID[req] = device_id;
    FILE_TYPE[device] = .FILE_TYPE[req] = TERMINAL_TYPE;
    IN_BLOCK_SIZE[device] = OUT_BLOCK_SIZE[device] =
        .FILE_BLOCK_SIZE[req] = 4 * .BYTES_PER_WORD;
    DROPPED[device] = SPURIOUS[device] =
        .FILE_LAST_BYTES[req] =
        .FILE_LAST_BLOCK[req] =
        .FILE_NEXT_BLOCK[req] = 0;

    return( .OK );
  }
```

This routine is called when a create instance request is received for an EIA terminal interface. Note that the EIA interface shares several device dependant routines with the Axis multiplexor (mux) terminal interface.

```
Eia_restart( device[] )
   \ Initialize an eia teminal interface.
  {
    template    EIA_STATE;
    unsigned    base, chan;

    base = CRU_BASE[device];
    code( .MOV., "r12", base );
    code( .SBO., EIA_DTR );
    code( .SBO., EIA_RTS );
    code( .SBO., EIA_ENABLE );
  }
```

This routine initializes the EIA hardware. It is called when the device is created, and when the system recovers after a power failure.

```
Eia_write( word req[], word device[], word sender[] )
  {
    template    EIA_STATE, .IO_REPLY, .IO_REQUEST, .PD;
    unsigned    base, p{}, count;

    if( EIA_WRITER[device] ) return( .BUSY );

    p = &.IO_BUF[req];
    count = .IO_BUF_LEN[req];
    base = CRU_BASE[device];
    code( .MOV., "r12", base );
    code( .TB., EIA_XMTING );   \ Test if transmission in progress
    code( .JEQ., setup );
output_char:
    code( .LDCR., 8, "ria6" );
    --count;
setup:
    if( count )
      {
        .REPLY_CODE[req] = .OK;
        WRITER_ID[device] = .ID[ EIA_WRITER[device] = sender ];
        OUT_BUF[device] = p;
        OUT_BUF_END[device] = p + pun( unsigned[], count );
        .STATE[sender] = WRITING_DEVICE;
        .BLOCKED_ON[sender] = pun( unsigned, device );
        return( .NO_REPLY );
      }
    else return( .OK );
  }
```

This routine is called in response to a .WRITE_INSTANCE request on an EIA interface.

```
Eia_read( word req[], word device[], word sender[] )
  {
    template    EIA_STATE, .IO_REPLY, .IO_REQUEST, .PD;

    if( OUT_PTR[device] == IN_PTR[device] )
      {
        if( EIA_READER[device] ) return( .BUSY );
        READER_ID[device] = .ID[ EIA_READER[device] = sender ];
        .STATE[sender] = READING_DEVICE;
        .BLOCKED_ON[sender] = pun( unsigned, device );
        return( .NO_REPLY );
      }
    Read_circular( req, device );
    return( .OK );
  }
```

This routine is called in response to a .READ_INSTANCE request on an EIA interface.

```
unsigned Mux_release( word req[], word device[], word sender[] )
  {
    template    MUX_CHAN_STATE;

    USED[device] = .FALSE;
    Free_buffer( IN_BUF[device] );

    return( .OK );
  }
```

A simple routine to release storage when device is removed, in response to a .RELEASE_INSTANCE request.

```
Mux_read_clear( device )
  {
    template    MUX_CHAN_STATE;

    MUX_READER[device] = 0;
  }
```

Called when a process reading an EIA interface is destroyed to inform the device handler.

```
Mux_write_clear( device )
  {
    template    MUX_CHAN_STATE;

    MUX_WRITER[device] = 0;
  }
```

Called when a process writing an EIA interface is destroyed to inform the device handler.

Bibliography

1. Cheriton, D. R. Designing an Operating System to be Verifiable. Technical Report 79-9, University of British Columbia, Vancouver, 1979.

2. Lockhart, T. W. The Design of A Verifiable Operating System Kernel. Technical Report 79-15, University of British Columbia, Vancouver, 1979. (based on M.Sc. thesis University of British Columbia, 1979).

3. Cheriton, D. R. Verex Servers. Technical Report 81-??, University of British Columbia, Vancouver, 1981.

4. Cheriton, D. R. Multi-process structuring and the Thoth operating system. Technical Report 79-5, University of British Columbia, Vancouver, 1979. (based on Ph.D. thesis University of Waterloo, 1978).

5. Ritchie, Dennis M. The UNIX I/O System. Bell Laboratories, 1975

6. Brinch Hansen, P. The nucleus of a multiprogramming system. Communications of the A.C.M. 13, 4 (April 1970).

7. Falk, Gilbert. The Structure and Function of Network Protocols. Computer Communications Wushow Chou, ed. Prentice-Hall, Inc., 1983.

8. Ritchie, The UNIX Time_Sharing System Communications of the A.C.M. 17, 7 (July, 1974).

9. Deering, Stephen E. Multi-Process Structuring of X.25 Software. Technical Report 82-11, University of British Columbia, Vancouver, 1982.

10. Cheriton, D. R. Distributed I/O using an Object-based Protocol. Technical Report 81-1, University of British Columbia, Vancouver, 1981.

11. Dijkstra, E. W. The Structure of the "THE" Multiprogramming System. Communications of the A.C.M. 11, 5 (May 1968).

12. Madnick, Stuart E. Design Strategies for File Systems. Project MAC, TR 78, Massachusetts Institute of Technology, Massachusetts, 1980.