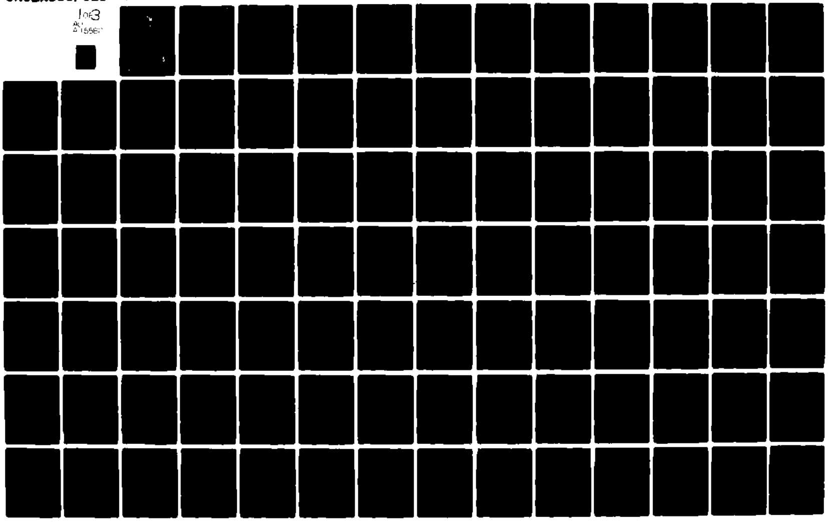
AD-A115 582   AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
A UNIX BASED DEVICE DRIVER FOR THE VECTOR GENERAL 3404 GRAPHICS--ETC(U)
MAR 82 B R STEWART
UNCLASSIFIED AFIT/GCS/MA/81D-6 NL

1 of 3
AD
A115582



0

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY (ATC)
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/MA/81D-6

1

A UNIX BASED DEVICE DRIVER FOR THE VECTOR GENERAL 3404 GRAPHICS DISPLAY SYSTEM

THESIS

AFIT/GCS/MA/81D-6

Bradley R. Stewart
2nd Lt USAF

AFIT/GCS/MA/81D-6

A UNIX BASED DEVICE DRIVER FOR THE VECTOR GENERAL 3404 GRAPHICS DISPLAY SYSTEM

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by
Bradley R. Stewart, B.S.
2nd Lt USAF
Graduate Computer Systems

March 1982

Approved for public release; distribution unlimited

Preface

The purpose of this study was to develop device driver software for Vector General 3404 Graphics Display System. The device driver software was installed on a PDP11/60 computer running under the UNIX version seven operating system.

This report discusses all of the major components of the sys‿em. These include the UNIX peripheral device I/O processing routines, the hardware interface between the PDP11/60 and the display system, the Vector General 3404 Graphics display system, and the device driver routines. I believe this work will be very helpful to anyone working on peripheral device I/O processing under the UNIX operating system.

I would like to thank my advisor, Professor Charles W. Richard, Jr., for his constant support and encouragement during this study. Deep gratitude is also expressed to Dr. J. Lions of the University of New South Wales for his brilliant commentary on the UNIX operating system. And finally, I wish to acknowledge my gratitude to Mary Minnick for her effort in typing this thesis.

Bradley R. Stewart

Contents

Contents

Contents

Contents

Contents

List of Figures

List of Tables

Abstract

A device driver for the Vector General 3404 Graphics Display System was installed under the UNIX version seven operating system on a PDP11/60 computer. This was accomplished by modifying an existing device driver which was designed to run under version six of the UNIX operating system.

The major topics addressed in this report are the C programming language, peripheral device I/O processing under UNIX, the hardware interface between the PDP11/60 and the graphics display system, the graphics display system itself, and the existing device driver software.

Structure charts were used to document the design of the UNIX peripheral device I/O processing software and the design of the device driver software. Modifications to the original device driver were easily accomplished due to the top-down modular design of the original software. UNIX provided a straight-forward interface for adding the device driver software to the system.

A UNIX BASED DEVICE DRIVER FOR THE VECTOR GENERAL 3404 GRAPHICS DISPLAY SYSTEM

I Introduction

The problem addressed in this thesis investigation was the development and installation of device driver software for a vector General 3404 Graphics Display System (hereafter referred to as the VG graphics device or the VG display system).
The software was installed under the UNIX version seven operating system running on a PDP11/60 computer. This effort was intended as a first step in the development of a high-level interactive graphics system for the PDP11/60 running under UNIX.

This chapter presents the background to the problem, scope and objectives, approach taken, conventions used, and finally an overview of the remainder of the thesis.

Background

In July, 1981 the RSX-11M operating system running on the PDP11/60 computer at the Air Force Institute of Technology (AFIT) was replaced by the Bell System's UNIX version seven operating system. This replacement was justified by a number of desirable UNIX features not offered by the RSX-11M operating system.

First of all, UNIX provides more software tools for education. It supports several different programming languages and provides excellent facilities for document preparation.

UNIX also provides very powerful tools for program development. An example is the UNIX capability of creating a "pipe" for inter-process communication. In a pipe, as output is generated from one program it is immediately made available as input to the next program (Refs 2:2; and 13:8). Therefore, a pipe facilitates executing programs together as a complete system. That is, a large software system can be designed and developed in small pieces, then brought together and executed in a pipe.

The UNIX system is totally self supporting. All UNIX software is maintained on the system. With only 10,000 lines of code, the system can easily be understood and maintained by one person. Of the 10,000 lines of UNIX code, less than ten percent is written in assembly language. The remaining ninety percent is written in the general-purpose procedural language "C". This high level language enhances system understandability and maintainability.

Ritchie and Thompson list the following desirable UNIX features seldom found even in larger operating systems (Ref 13:1).

1. A hierarchical file system incorporating demountable volumes.
2. Compatible file, device, and inter-process I/O.
3. The ability to initiate asynchronous processes.
4. System command language selectable on a per user basis.
5. Over 100 subsystems, including a dozen languages.
6. High degree of portability.

These features make UNIX simple, elegant, and easy to use.

With the upgrade to UNIX, it became necessary to upgrade the graphics package on the PDP11/60. FGP34, a graphics package based on ACM/SIGGRAPH's Core System standard proposal (Ref 15), is the package that ran under the RSX-11M operating system. This package is not readily compatible with UNIX. In order to run FGP34 under UNIX, a new device driver would have to be written. This could be very difficult to impossible depending on how strongly the FGP34 software is dependent on the RSX-11M operating system. Another issue that must be considered is that FGP34 does not support complete device independence. That is, it only runs with the Vector General 3400 series display systems. Therefore, when other types of graphics devices are installed on the PDP11/60 in the future, the FGP34 will not support them.

As a result of these problems and limitations, it was decided to acquire a better graphics software system, e.g., one based on the Core System that is operating system independent and device independent. One good candidate that has been identified is GRAFLIB, a graphics software system developed by the Lawrence Livermore Laboratory (Ref 6).

No matter which high level graphics software system is finally implemented, a new low level device driver had to be installed under UNIX for the VG graphics device. When this investigation was begun, the author knew of no VG device drivers written to run under UNIX version seven. At the same time, it was known that two different VG device drivers did exist for UNIX version six. One had been developed at The University of Kansas. Another had been developed at The University of Texas at Austin. In order not to "re-invent" the wheel, it was decided to modify one of these existing drivers to run under UNIX version seven.

The driver developed at the University of Texas at Austin was chosen because of its straight forward, top-down design and because the driver source code was easy to obtain. The driver was written by Douglas McCallum in support of his thesis on machine-independent interactive computer graphics (Ref 12). It was designed for the UNIX version six operating system running on a PDP11/34 computer.

Scope and Objectives

This thesis investigation was devoted to updating and installing McCallum's VG device driver on the PDP11/60 computer under the UNIX version seven operating system.

One main objective was to, as much as possible, use McCallum's device driver software "as is". Modifications were only made to make the driver compatible with UNIX version seven and to meet the space limitations of AFIT's PDP11/60 computer. Also, since McCallum's driver did not support the VG's data tablet input device, software was developed and incorporated to support AFIT's data tablet.

Another main objective was to document how the VG driver works. This included an explanation and description of the UNIX operating system, the hardware interface between the PDP11/60 and the VG graphics device, the VG graphics device itself, and the driver routines.

Approach

This project required a working knowledge of the "C" programming language, the UNIX operating system (both versions six and seven), the PDP11/VG3404 hardware interface, the VG graphics device at the register level, McCallum's driver software, and driver installation procedures.

First, UNIX was studied from a user's point of view to learn how to use the system. The article "An Introduction to the UNIX Shell" and "UNIX for Beginners - Seventh Edition" served as tutorials for this step (Refs 2 and 9). The UNIX text editor was learned next by studying the article "A Tutorial Introduction to the UNIX Text Editor" (Ref 8).

Next, the "C" programming language was learned by writing and executing programs that illustrated the major features of the language. Kernighan and Ritchie's book entitled The C Programming Language was used as a tutorial during this step (Ref 7). This step was essential since both UNIX and the device driver are written in "C".

After learning the basics above, UNIX was studied from a systems point of view to learn how it deals with device drivers in general. J. Lions' commentary on the UNIX operating system, along with listings of UNIX source code, served as the main tutorial for this step (Refs 10 and 11). The differences between UNIX version six and UNIX version seven were also studied at this point.

Study of Vector General documentation provided an understanding of the PDP11/VG3404 hardware interface and the VG graphics device at the register level. The most important of these documents were the Programming Concepts Manual, the System Reference Manual, the PDP11 Interface Specification, and volume one of The Series 3400 Technical Manual (Refs 17-20).

The knowledge obtained from the above studies helped the author understand McCallum's driver software. Some help was also received through telephone conversations with Douglas McCallum. Once the driver was understood, it was updated to run under UNIX version seven. Next, the driver was installed on the system using the articles "Regenerating System Software" and "Setting Up UNIX" as a guide for installation procedures (Refs 4 and 5).

The final step was the development and incorporation of routines to handle the VG data tablet input device. McCallum's routines for the VG function switches and keyboard input devices served as a guide for writing these routines.

Conventions Used

A few conventions are identified here that are used throughout the report.

All references to UNIX system commands are specified by the command name followed by a section number in parenthesis. The section number refers to the section of the UNIX Programmer's Manual (Ref 1) where the command is defined.

For example, cp(1), refers to the copy system command which is found in section 1 of the UNIX Programmer's Manual. This system was adopted because the UNIX Programmer's Manual does not have page numbers.

Another convention that merits explanation is how computer code is cited in this report. Two types of code are cited in this report; a stream of UNIX system commands entered from a terminal and listings of "C" language statements taken from computer programs. The following UNIX system command stream illustrates how a stream of UNIX command statements

```
1.  # cd /sys/conf
2.  # cp /sys/dev/vg.c/sys/dev/vg
3.  # mkdev_i vg
4.  cp ../h/param_i.h ../h/param.h
5.  a - vg.o
6.  # rm /sys/dev/vg
7.  #
```

is cited in this report. First, the statements are numbered sequentially to provide a means of referencing each individual line. If only one statement is cited then it is not numbered. The symbol "#" is a prompt sign printed by the system. Following the prompt sign the user types a command statement followed by a carriage return. The system executes the command then prints another "#" to prompt the user for more input. Lines not beginning with the "#" prompt, as with lines four and five above, represent messages or text printed during the execution of a command.

The "#" prompt is also an indication that the user is logged in as the super-user. The super-user is granted special access rights and priviledges that other users do not receive. These rights and priviledges allow the super-user to make any necessary changes to the system, such as install a new device driver.

The "C" language statement listing

```
1.  dtclose()
2.  {  extern struct cdevsw vgdev[];
3.         POUT(dtb, 0);
4.         while (getc(&vgunit[1].io)  >= 0);
5.  }
```

is an example of how portions of computer programs are cited in the report. The statements are simply numbered sequentially so each individual line can be referenced easily.

Overview of the Thesis

The block diagram depicted in Figure 1 orients the reader to the system components and the communication paths between them. The main body of the thesis describes these components and communication paths in detail. The remainder of the report is outlined below.

In order to establish a common base to work from, some basic concepts of the UNIX operating system are presented in chapter two. This includes a description of the UNIX File System, the UNIX I/O System, and process management.

Chapter three describes in detail how UNIX processes user program requests for peripheral device I/O, how it deals with device drivers, and how it processes interrupts from peripheral devices.

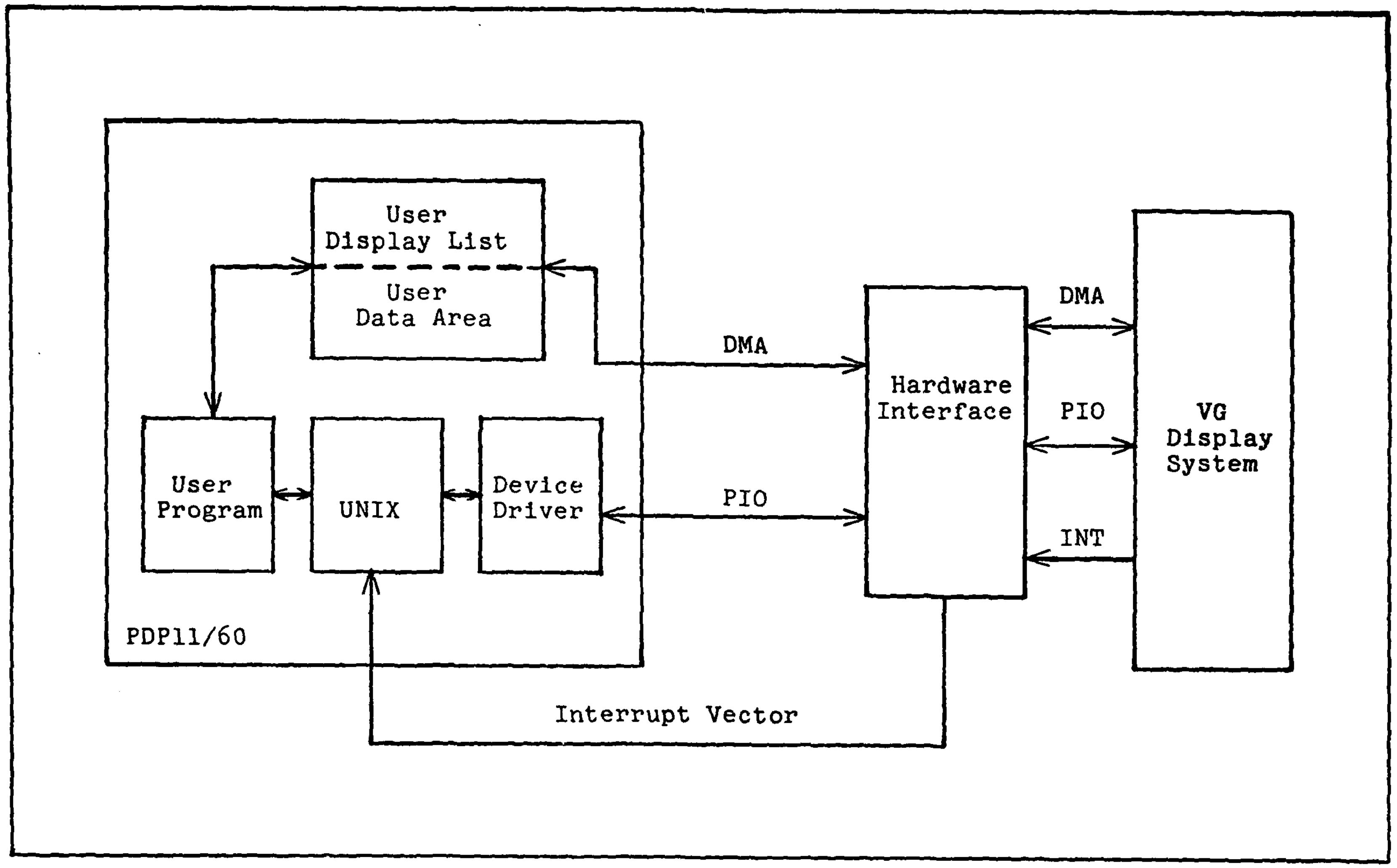


Fig 1. Organization of Major System Components

The VG is described down to the register level in chapter four. This is necessary because the device driver deals mainly with reading and writing the VG's internal registers.

Chapter five is a description of the hardware interface between the PDP11/60 and the VG graphics device. All data and control communication between the PDP11/60 and the VG take place via this interface.

The device driver software is documented in chapter six. This includes a specification of the overall requirements, a description of the software design, an a detailed discussion of implementation details. The discussion on driver implementation includes both user level implementation and documentation of the driver routines in their final state, e.g., after updating for UNIX version seven and trimming to meet space limitations.

Chapter seven describes the changes made to McCallum's original driver to make it compatible with AFIT's system.

The procedure for installing the device driver is described in detail in chapter eight.

The software testing methodology is described in chapter nine. All of the tests performed on the driver software are also included.

Finally, conclusions and recommendations are given in chapter ten.

II Preliminary Concepts

A detailed knowledge of certain aspects of the UNIX operating system is required for developing and installing peripheral device driver software on the system. In order to gain this detailed knowledge, the basic concepts must first be understood.

This chapter presents a discussion of some basic UNIX concepts. Emphasis is placed on those concepts that will aid the reader in understanding the more detailed UNIX concepts presented in subsequent chapters. The main ideas covered here are the UNIX file system, the UNIX I/O system, and process management.

The UNIX File System

Ritchie and Thompson have stated, "the most important role of the system is to provide a file system" (Ref 13:2). UNIX supports a hierarchical disk based file system composed of three different kinds of files: ordinary, directory, and special. Each of these files is stored as a one dimensional array of bytes. Structure within these files is controlled by the programs that use them and not by the system.

Ordinary files, directory files, special files, file system hierarchy, file path names, and file system implementation are discussed in this section.

Ordinary Files. Ordinary files can be created by any user. They contain whatever the user puts in them, e.g., data, source programs, object (binary) programs, etc. Access permission to ordinary files is controlled by the file owner and/or by the super-user, i.e., the person in charge of maintaining the entire system.

Directory Files. Directory files are maintained by the system. They can only be written by the system. A user program may not open a directory file for writing. Directories may contain names of ordinary files, special files, and other directory files. For each file name entry, the directory maintains a pointer, called the i-number (for index number), to the information actually describing the file. In other words, each directory entry provides a mapping between a file name and the actual file. The i-number will be described later in detail.

Special Files. A special file is a file that has been associated with an I/O device. By UNIX convention, these files all reside in directory /dev. User programs access I/O devices through references to the special files associated with the I/O devices. User programs may open, close, read, and write special files as if they were ordinary disk files. When special files are referenced from a user program, the system calls the appropriate device driver routine to activate the associated device (Ref 13:3). This is a key concept in this thesis because the VG graphics device has four of these special files associated with it for I/O purposes. These four special files are described in detail in chapter six.

File System Hierarchy. Directories are maintained by

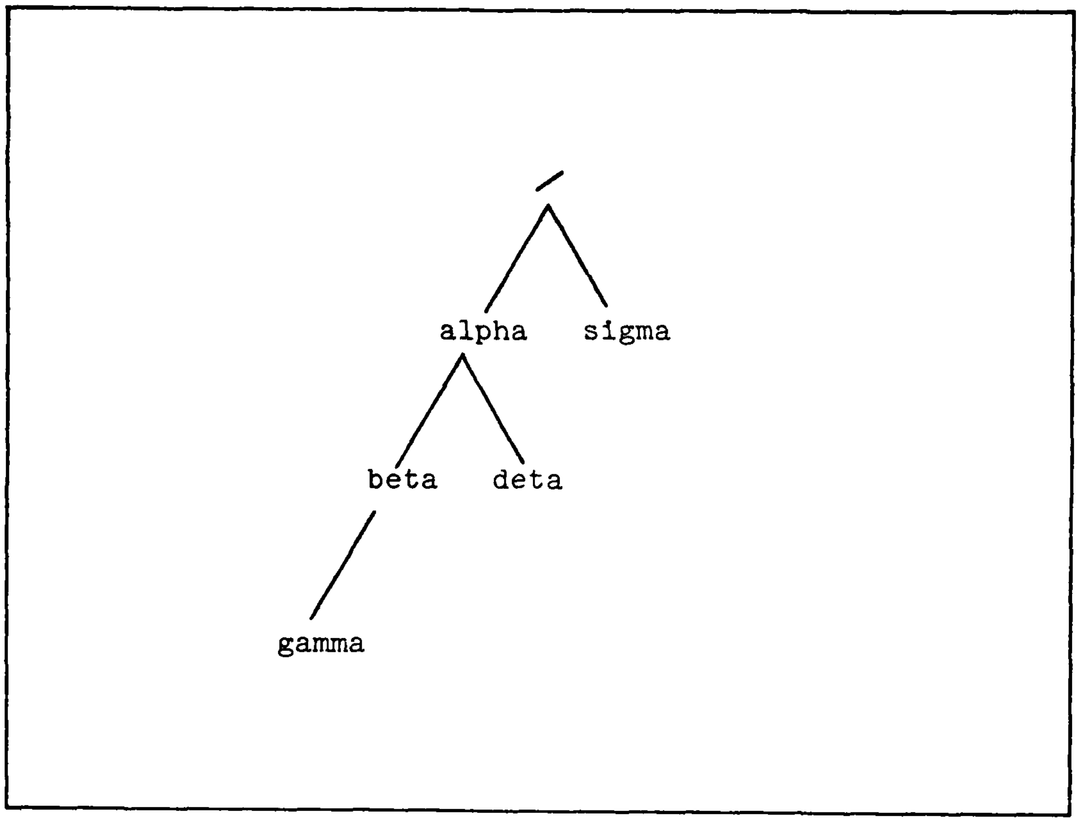


Fig 2. File System Hierarchy

the system as a hierarchy in the form of a rooted tree. An example of such a rooted tree is illustrated in Figure 2. In this figure, as with the UNIX file system, the root directory is denoted by a slash character, "/". The root directory contains the files alpha and sigma. Alpha is also a directory file containing files beta and delta. Beta is a directory file containing file gamma. All non-leaf files in the rooted tree are directory files. Files delta, gamma, and sigma could be either ordinary, special, or empty directory files.

Many of the higher level directories in the file system hierarchy are reserved for system use. They contain system commands, UNIX source and object files, utility programs, etc.

The super user adds directories to the file system for each user. These directories may be added to any level of the hierarchy. They are created for the user's own files. Users manage files within their respective "home" directories through the use of system calls. They may do such things as add and remove files from their own directories. They may also create and manage sub-directories attached to their original home directories.

File Path Names. A file may be specified to the system in terms of its path name. Ritchie and Thompson describe this concept well.

> "When the name of a file is specified to the system, it may be in the form of a path name, which is a sequence of directory names separated by slashes, "/", and ending in a file name. If the sequence begins with a slash, the search begins in the root directory. The name /alpha/beta/gamma causes the system to search the root for directory alpha, then search alpha for beta, finally to find gamma in beta. Gamma may be an ordinary file, a directory, or a special file. As a limiting case, the name "/" refers to the root itself." (Ref 13:3)

If the path name does not start with a "/" then the system begins searching in the user's current directory. When a user logs onto the system, the user's assigned home directory becomes the current directory. The current directory may be changed through use of the change directory system call, cd(1).

File System Implementation. A detailed description of the implementation of the UNIX file system is given by Thompson and Ritchie (Ref 13:6-7 and 16:7-9). The main ideas are presented here.

The system maintains a list of file definitions called the "i-list". This list resides on secondary storage (usually on disk) and consists of one "i-node" for each file that exists in the file system. The integer offset of an i-node in the i-list is called the i-number. It is used for referencing the i-node and is stored in a directory along with the file name associated with the i-node.

Each i-node contains all the information needed to define a file, such as; the type of file, access permissions, the number of links to the file, etc. (Ref 13:6) For non-special files, the i-node contains information about where the file resides on disk (Refs 13:6 and 16:7). For special files, the i-node contains a device class and a device name. These are used by the system to invoke the appropriate device driver when a user program requests access to a special file.

Figure 3, adapted from Thompson (Ref 16:8), shows the data structures maintained by the system during file access. Each user process is allocated an open file table. This table contains pointers to entries in the system open file table. As a user process is swapped in and out of core, its open file table is swapped along with it. The system open file table and the active i-node table are always resident in core. The i-list resides on disk.

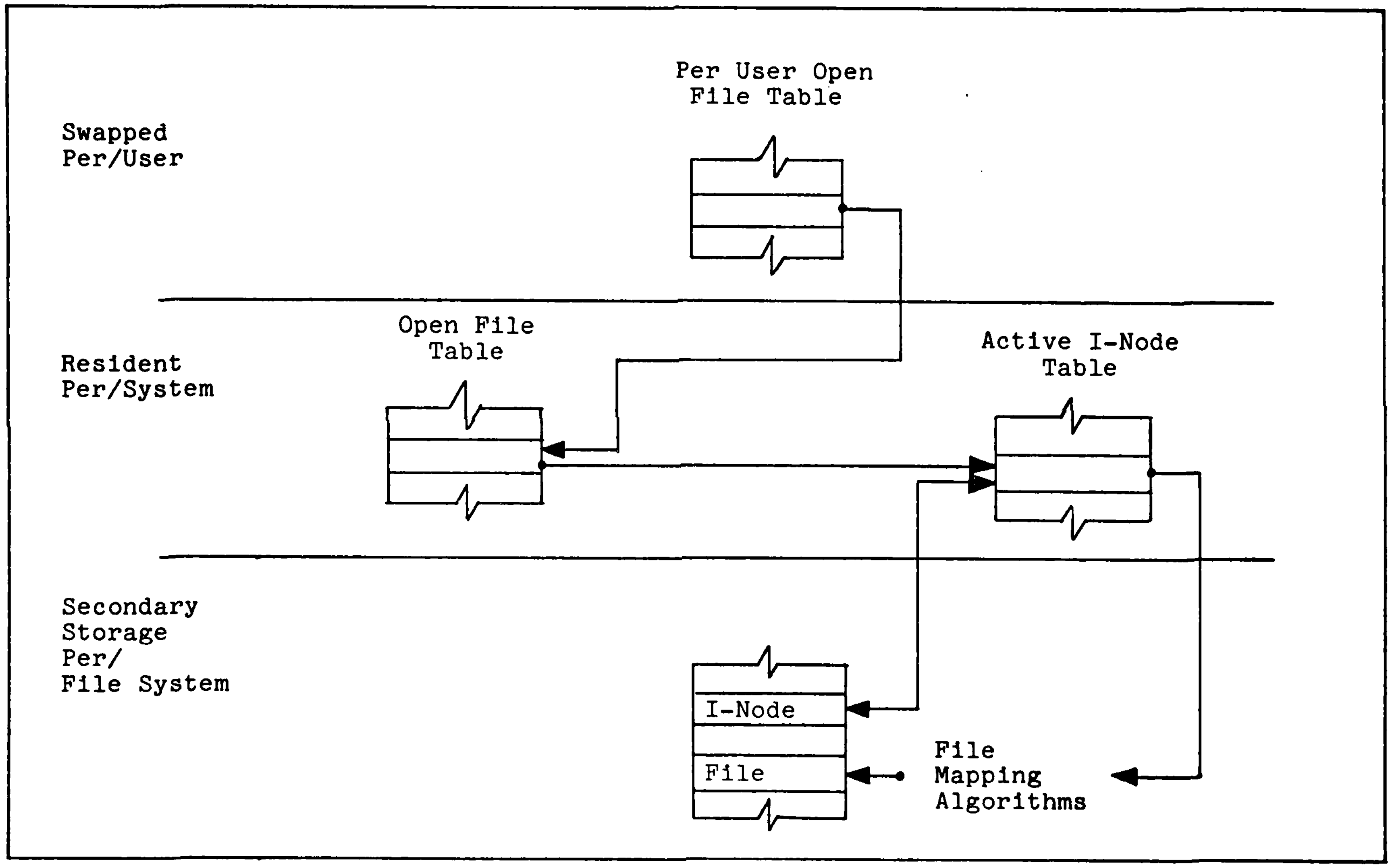


Fig 3. File System Data Structure

To access an existing file, a user program must first "open" it via the open(2) system call. The file's pathname is specified as one of the input parameters for this call. The system uses the pathname to search the hierarchy of directories until the specified file name is found. Next, the i-number stored in the directory with the file name is retrieved. The i-number is used to access the appropriate i-node. The system checks the file's access permissions (stored in the i-node) to verify that the requested access is legal. If it is, the system copies the disk version of the i-node into the active i-node table. A pointer to this active i-node table entry is entered in the system open file table. A pointer to this system open file table entry is entered in the user's open file table. The integer offset of the entry just made in the user's open file table is called a file descriptor. It is passed back to the user program. The user program passes the file descriptor as an input parameter on all subsequent system calls requesting access to the open file.

The UNIX I/O System

This section begins with a description of basic I/O system calls. This is followed by a discussion of device classes and device names which are used during peripheral device I/O processing.

Basic I/O System Calls. A user program requests I/O through the use of the open(2), close(2), read(2), write(2), stty, and gtty system calls (see ioctl(2) for the stty and gtty system calls). The open(2), close(2), read(2), and write(2) system calls may be used on both ordinary and special files, while the stty and gtty calls are only used on special files. The open(2) call opens a file for access while the close(2) call terminates access to a file. The open(2) call passes two parameters to the system; (1) the path name of a special file and (2) an access mode. If the access mode specified is 0 then the I/O request is for reading only. If the access mode equals 1 then the request is for writing only. If it equals 2 then the request if for both reading and writing. The open(2) call returns a file descriptor which must be used in subsequent I/O requests on the open file. The close(2) system call passes one parameter to the system; a file descriptor.

The read(2) and write(2) calls pass three parameters to the system; (1) the file descriptor obtained from the open(2) system call, (2) a pointer to a user buffer, and (3) the number of bytes requested. With the read(2) system call, up to the number of bytes requested are read into the user buffer. The system returns the number of bytes actually read. With the write(2) command, the number of bytes requested are written from the user buffer to the specified file. The number of bytes actually written is returned to the user program.

The stty and gtty commands are used to set and get characteristics of peripheral devices. These system calls each pass two input parameters to the sytem; (1) a file descriptor and (2) a pointer to a user buffer. The contents of the user buffer specify which device characteristics to set or get.

When a user program invokes one of these basic I/O system calls on a special file, the operating system activates the associated peripheral device via device driver routines. The device class, part of the device name, and the type of I/O system call determine which device driver routine is invoked. The appropriate device driver routine performs the requested I/O function on the peripheral device then returns control to the operating system which then returns control to the user program, passing back the appropriate data. Device classes and device names are now discussed.

Device Classes. Each I/O device falls into one of two categories; block oriented or character oriented. Block oriented devices are devices such as disk and tapes which deal with 512-byte blocks. All other devices are considered charater oriented. Therefore, the VG graphics device is a character oriented device.

Special files associated with block I/O devices are marked as block oriented, while those associated with character devices are marked as character oriented. This information is carried in each special file's i-node.

Device Names. The system assigns a device name to each special file. It is stored in the special file's i-node. The device name is made up of a major device number and a minor device number. These are stored in the i-node as a 16 bit computer word with the major number in the high order 8 bits and the minor number in the low order eight bits (Ref 14:1).

When a user program requests access to a special file, the file's device class, major device number, and the type of I/O request determine which device driver routine to invoke. The special file's minor device number is passed to the device driver routine as an argument (Ref 16:5).

Any meaning associated with the minor device number is assigned by the device driver routine itself. For example, if there are several identical I/O devices on a system, the minor device number could be used to indicate which one of the I/O devices to activate. Another example would be I/O devices composed of several sub-devices. In this case, the minor device number could be used to indicate which sub-device to activate.

Process Management

Ritchie and Thompson identify an "image" as a computer execution environment and a "process" as the execution of an image (Ref 13:8). Roughly speaking, a process may be defined as "a program in execution" (Ref 10:7-1).

UNIX allocates two data structures for each process on the system. They are the "proc" structure and the "user" structure. These structures make up part of the overall process image. A complete listing of each is included in Appendix A.

The Proc Structure. The proc structure for each process is permantly resident in core. This structure is defined in the UNIX source file /sys/h/proc.h. It contains information that must be accessible at any time, especially when the main part of the process image has been swapped out to disk. Lions describes the information carried in the proc structure in his commentary on the UNIX operating system (Ref 10:7-2).

The User Structure. The user structure assigned to each process is swapped in and out of core with the swapable portion of the process image. At any given time, the only user structure in core is the one assigned to the process currently being executed. While in core the user structure is referenced as the "u" structure.

The u structure is defined in the UNIX source file /sys/h/user.h. It contains such information as user identification, parameters for I/O operations, file access control, system call parameters, and accounting information.

The u structure is accessed often during execution of a process. Each element of the u structure is accessed by stating the name of the structure, followed by the structure member operator '.', followed by the element name (Ref 7:120). For example,

```
u.u_base
```

is a reference to the element u_base of the u structure.

Both the UNIX operating system and the device driver routines access the u structure often while processing peripheral device I/O requests. The individual elements of the u structure needed for I/O processing are described throughout this report as needed.

Summary

Some basic concepts of the UNIX operating system were presented in this chapter. Emphasis was placed on those UNIX concepts that pertain to this thesis project. With these basic concepts as a foundation, the next chapter describes how UNIX processes user program requests for peripheral device I/O.

III <u>Peripheral Device I/O</u>

The UNIX operating system is the focal point for all peripheral device I/O processing. This chapter is a discussion of how UNIX processes user I/O requests and peripheral device interrupts. Emphasis is placed on character oriented peripheral devices. This will help the reader to understand how UNIX deals with the VG graphics device.

This chapter is divided into three sections. The first presents a high level discussion of the flow of control during I/O processing. This is intended to orient the reader to the overall role of the UNIX operating system in peripheral device I/O processing. The next section describes how user program I/O requests are processed. The last section describes how peripheral device interrupts are processed.

<u>Flow of Control During I/O Processing</u>

UNIX controls the processing of all user I/O requests and all peripheral device interrupts. The block diagram in Figure 4 illustrates the overall flow of control during I/O processing. When a user program requests I/O on a peripheral device, control is transfered to UNIX. First, UNIX executes the device independent routines needed for the I/O request, then it determines which device driver routine to invoke for the required device dependent processing. Next, the appropriate device driver routine is called. It performs the requested I/O function then returns control to UNIX. UNIX

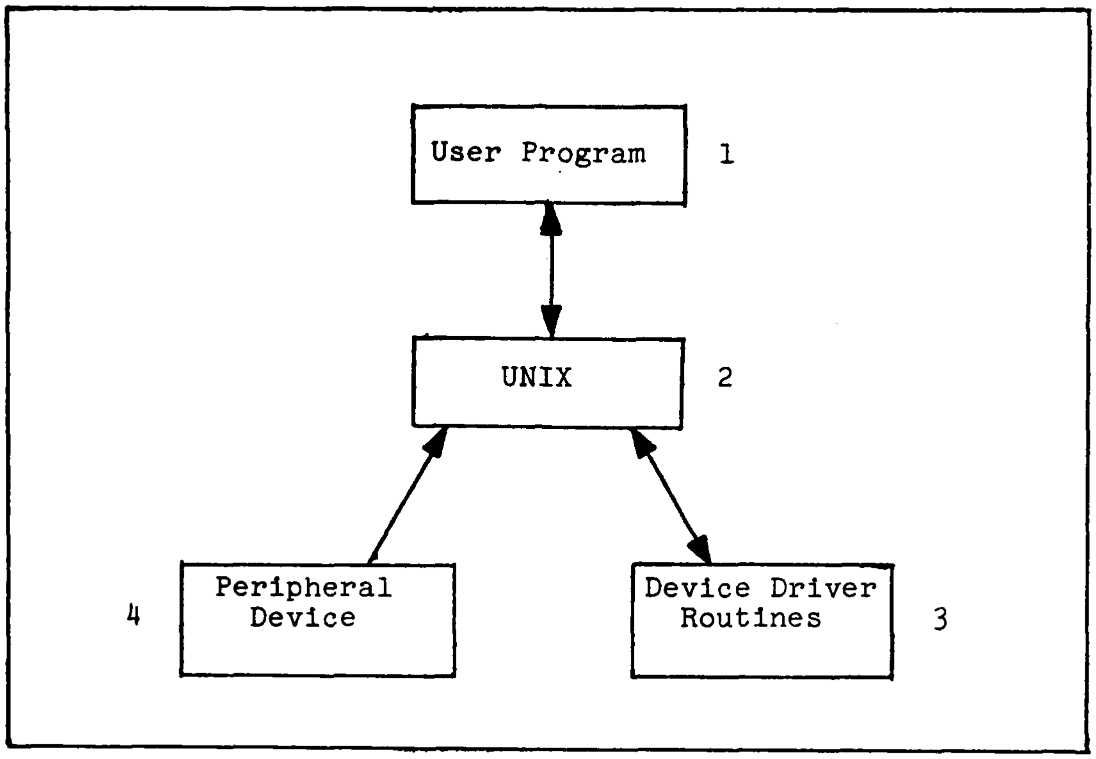


Fig 4. Flow of Control During I/O Processing

finishes processing the I/O request then returns control to the user program. In terms of the components of Figure 4, the typical flow of control for processing a user I/O request is 1,2,3,2,1.

When a peripheral device signals an interrupt to the PDP11 processor, control is transfered to the interrupt vector in UNIX. The interrupt vector first transfers control to the UNIX assembly language interrupt handler which performs device independent interrupt processing. Next, the device dependent interrupt handler, which is part of the device driver software, is invoked. The device dependent interrupt handler processes the interrupt then returns control to UNIX. UNIX returns con-

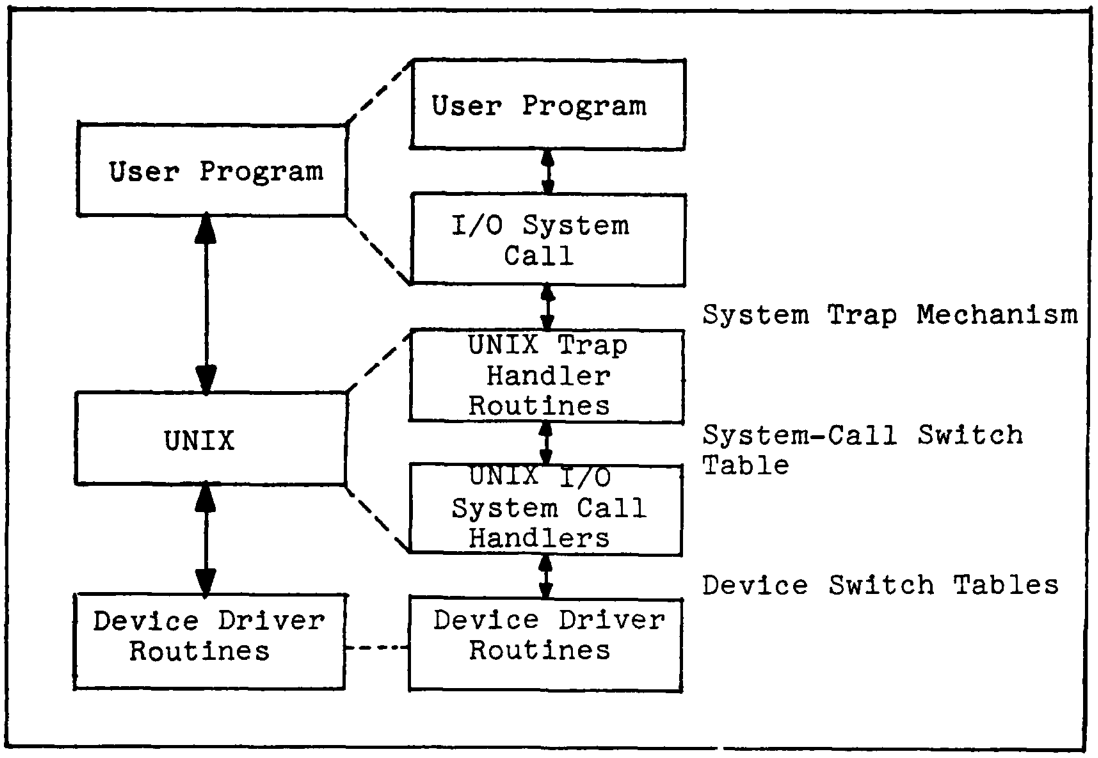


Fig 5. I/O Processing Routines and Control Transfer Mechanisms

trol to whoever had it at the time the interrupt occurred. In terms of Figure 4, the flow of control for processing a peripheral device interrupt is 4,2,3,2.

Processing User Program I/O Requests

The portion of Figure 4 dealing strictly with processing user I/O requests is expanded in Figure 5 to show more detail. This figure illustrates the groups of routines called to process an I/O request and identifies the mechanisms used to transfer control between each group of routines. Control is transfered from the I/O system call to the UNIX trap handler routines via the system trap mechanism; from the UNIX trap handler routines to the UNIX I/O system call handler routines via the system-call switch table; and from the UNIX I/O system call handler routines to the device driver routines via the device switch tables. The remainder of this section describes each group of routines and each switch mechanism starting with the user program and ending with the device switch tables. Much of this information is found in Lions' commentary on version six of the UNIX operating system (Ref 10:Chapters 9, 10, 11, 12, 15, 18, and 19). However, due to differences between UNIX versions six and seven some of the information presented here was obtained directly from the UNIX version seven source code. When this is the case, the appropriate UNIX version seven source file is referenced.

The device driver routines are not described in detail here. Chapter six is devoted to a detailed description of the VG device driver routines.

The User Program and I/O System Calls. A user program requests peripheral device I/O via the I/O system calls open(2), close(2), read(2), write(2), stty, and gtty (see ioctl(2) for stty and gtty). These I/O system calls each compile to a trap instruction followed by the call's input parameters listed in the order that they were specified in the call. For example, the system call

```
                    read(fildes, buffer, mode)
```

compiles to

```
                    trap 3
                    fildes
                    buffer
                    mode
```

The low order byte of the trap instruction is an integer system-call identifier which uniquely identifies which system call caused the trap (Ref 10:10-2). In the example above, the number 3 represents the system call identifier for a "read" system call. Later, the system call identifier is used as an index into the system-call switch table to fetch the address of the appropriate system-call handler routine.

The System Trap Mechanism. Traps occur as the result of events internal to the CPU (Ref 10:9-3). Several different classes of system events cause the CPU to trap. Some of the different classes are bus errors, illegal instructions, power failure, execution of a system call trap instruction, etc. (Ref 10:9-3). A trap vector exists for each different class of events. All of the trap vectors are defined in the source file /sys/conf/l.s. The version of this file used when the VG graphics device is configured on the system, /sys/conf/l.s.vg, is listed in Appendix B.

When a system event causes a trap to occur, the CPU immediately transfers control to the associated trap vector. This is the first step for processing the trap. The trap vector associated with system calls is illustrated in

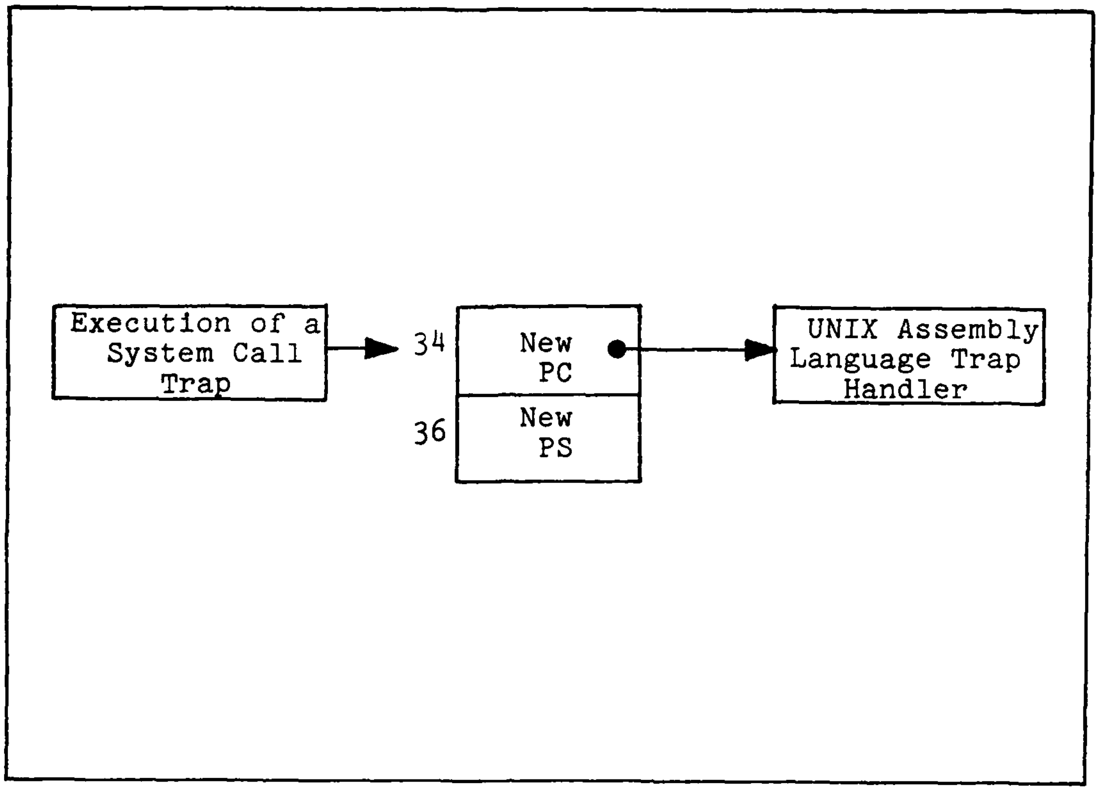

Execution of a System Call Trap
34
New PC
36
New PS
UNIX Assembly Language Trap Handler

Fig 6. System-Call Trap Vector

Figure 6. This trap vector begins at location 34 (octal) of low core (Ref 10:10-3). Initially, location 34 contains the assembly language "start" routine (see line 31, Appendix B). This is used when booting up the system, then location 34 is overlayed with the address of the assembly language trap routine. Location 36 contains the new processor status (PS) value to be used while handling the trap.

When the CPU executes a system call trap instruction, it immediately loads the program counter (PC) and the processor status (PS) word with new values taken from vector locations 34 and 36 respectively (Ref 10:10-3). The old PC and PS are automatically saved on top of the system stack.

The old PC value is pointing at the first word after the trap instruction, i.e., the first system call input parameter. Control is now transferred to the new address held in the PC, i.e., the address of the UNIX assembly language trap routine (Ref 10:10-3).

UNIX Trap Handler Routines. The UNIX trap handler routines consist of the assembly language trap routine located in source file /sys/conf/mch_i.s and the C language trap routine located in source file /sys/sys/trap.c.

When the assembler trap routine gets control, it first saves the new PS on top of the system stack. Lions states, "it is important to save the PS as soon as possible, before it can be changed, since it contains information defining the type of trap that occurred" (Ref 10:10-3). Next, the assembler trap routine saves important system registers on top of the stack so that they may be restored after the trap is processed. Finally, the C language trap routine is called.

First, the C language trap routine processes the parameters specified in the I/O system call. These parameters are fetched from the user program string in the following ways (Ref 10:12-2):

1. via the special register r0;
2. as a set of words embedded in the program string following the "trap" instruction;
3. as a set of words in the program's data area.

The open(2) system call parameters are passed from the user program using method 2 above. That is, the two parameters specified in the open(2) call are picked up from the program string following the trap instruction. This is accomplished using the old PC value (fetched from the system stack) which is pointing at the parameter list. The parameters for the other five I/O calls are passed using a combination of methods 1 and 2 above. The first parameter of these five calls is placed in special register r0 when the trap instruction is executed. The remaining parameters are picked up from the program string following the trap instruction.

The C language trap routine fetches all the system call input parameters by first fetching the unique identifier for the system call from the low order byte of the trap instruction (Ref 10:12-2). This integer identifier is used as an index into the system-call switch table (described later) to retrieve two pieces of information; the total number of parameters required for the system call and the number of those parameters that were passed through special registers.

After fetching all the parameters, the C language trap routine places them in the argument array, u.u_arg[] , so that they may be retrieved later by the UNIX I/O system call handler routines. Depending on which I/O system call is made, u.u_arg[] contains one of the following sets of system call parameters.

```
1.  For the open(2) system call:
        u.u_arg[0] = file pathname;
        u.u_arg[1] = access mode

2.  For the close(2) system call:
        u.u_arg[0] = file descriptor.

3.  For the read(2) and write(2) system calls:
        u.u_arg[0] = file descriptor;
        u.u_arg[1] = pointer to a user buffer;
        u.u_arg[2] = number of bytes to be read
                     or written

4.  For the stty and gtty system calls:
        u.u_arg[0] = file descriptor;
        u.u_arg[1] = pointer to a user buffer.
```

After the system call parameters are placed in the u.u_arg[ ] array, the C language trap handler calls the appropriate UNIX I/O system call handler routine via the system-call switch table.

System-Call Switch Table. The system-call switch table is defined in file /sys/h/sysent.h as an array of structures. The array is initialized in file /sys/sys/sysent.c. The following C code declares the array but does not dimension or initialize it.

```
1.  extern struct sysent {
2.                  char     sy_narg;
3.                  char     sy_nrarg;
4.                  int      (*sy_call)();
5.  } sysent[] :
```

Lines 1-4 define a structure named sysent which consists of three elements. The first element, sy_narg, is used to specify the total number of arguments needed for a particular system call. The element named sy_nrarg is used to specify the number of arguments passed through special registers such

Sysent Table			
index	sy_narg	sy_rarg	(*sy_call)()
⋮	⋮	⋮	⋮
3	3	1	read
4	3	1	write
5	2	0	open
6	1	1	close
⋮	⋮	⋮	⋮
31	2	1	stty
32	2	1	gtty
⋮	⋮	⋮	⋮

Fig 7. System-Call Switch Table

as r0. The last element, (*sy_call)(), is a pointer to a function that returns an integer value (Ref 7:114-116).

Line five declares an undimensioned array of sysent structures. The array is also named sysent, which may cause some confusion. The sysent array is initialized in file /sys/sys/sysent.c to logically appear as a table with one row for each system call existing on the system. Figure 7 shows the table entries for the I/O system calls. Notice that the three elements of the sysent structure map directly onto each row of the table, thereby providing a means of retrieving data from the table. The table is indexed by the system call identifier obtained from the low order byte of the system call

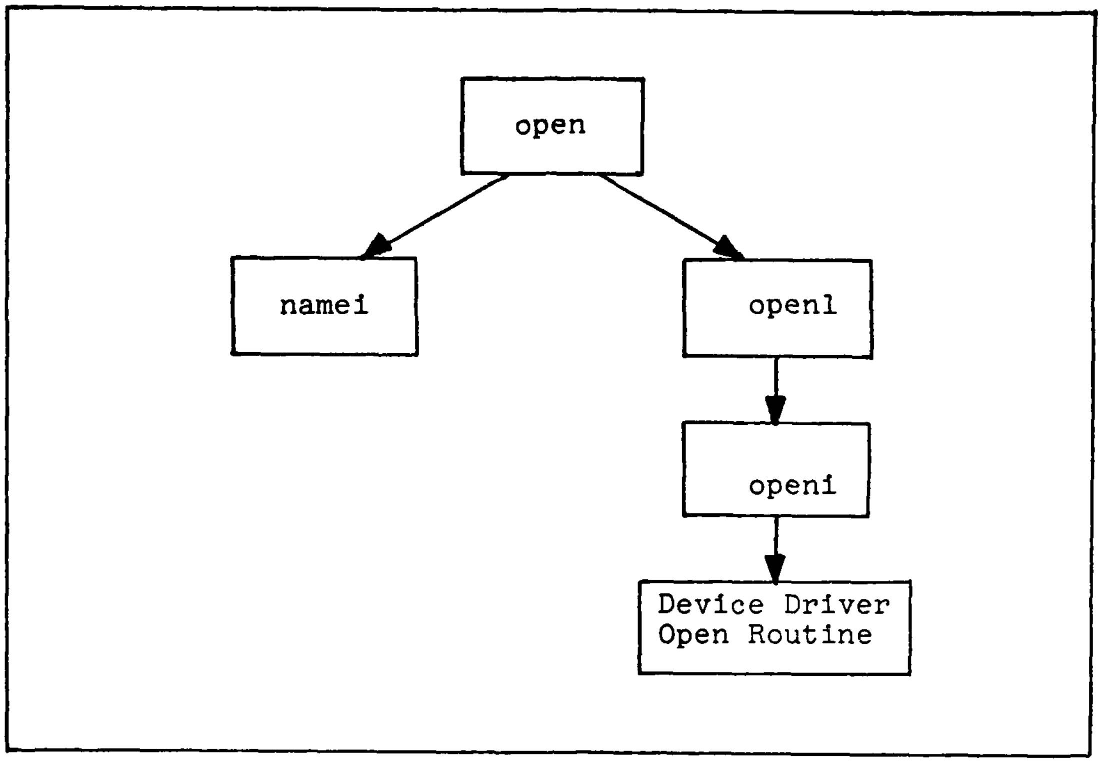


Fig 8. Routines for Processing an open(2) System Call

trap instruction. The first two columns of the table were used by the C language trap routine to determine how many parameters to fetch and how many of them were passed in special registers. The third column of the table, which contains the addresses of the I/O system call handlers, is used by the C language trap handler to call the appropriate I/O system call handler.

I/O System-Call Handler Routines. The I/O system calls open(2), close(2), read(2), write(2), stty, and gtty cause the C language trap handler to invoke the I/O system-call handler routines open, close, read, write, stty, and gtty. Each of these system-call handler routines is described later.

Open. Figure 8 illustrates the system-call handler routines invoked to process the open(2) system call. The open routine, located in source file /sys/sys/sys2.c, first calls the "namei" routine (located in /sys/sys/nami.c) to convert the file pathname (system call parameter 1 retrieved from u.u_arg[0]) into a pointer to an i-node. If the file has not been previously opened then namei makes a copy of the file's disk i-node in the active i-node table (Ref 10:18-3). This is accomplished via a call to the "iget" routine (Ref 10:18-3). Namei returns a pointer to the active i-node table entry. Next, open calls the "openl" routine passing it the pointer to the active i-node. Openl, located in source file /sys/sys/sys2.c, first checks file access permissions. Next it makes the appropriate entries in the system open file table and the user open file table. Finally, openl calls the "openi" routine. Openi, located in source file /sys/sys/fio.c, retrieves the special file's device class and device name. The device class indicates whether to call the driver open routine via the character device switch table or via the block device switch table. The major device number, taken from the high order byte of the device name, determines which device driver open routine to invoke via the device switch table. The appropriate device driver open routine is called with the minor device number (taken from the low order byte of the device name) passed as an argument.

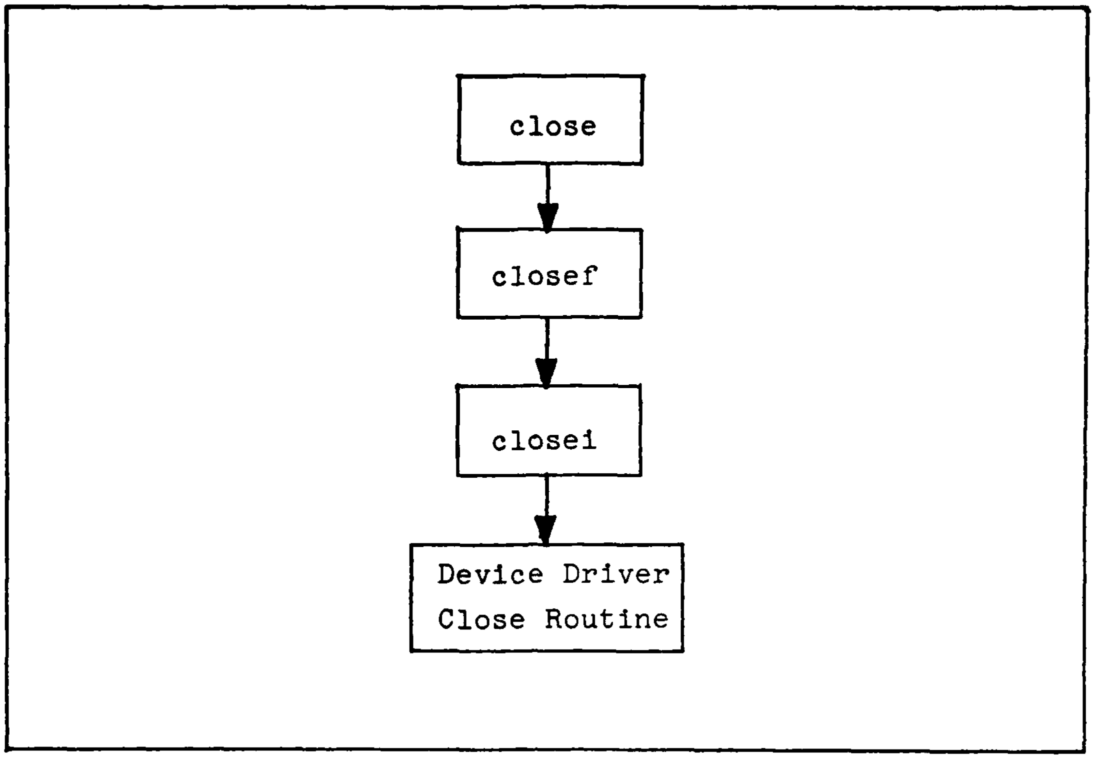


Fig 9. Routines for Processing a close(2) System Call

Close. Figure 9 illustrates the routines invoked to process a close system call. As stated by Lions, "the 'close' system call is used to sever explicity the connection between a user program and a file and thus can be regarded as the inverse of 'open'" (Ref 10:18-3).

The Close routine, located in source file /sys/sys/sys2.c, zeros out the appropriate entry in the open file table, u.u_ofile [], by fetching the file descriptor parameter from u.u_arg[0] and using it as an index into the open file table (Ref 10:18-4). Next, the close routine calls the "closef" routine. Closef, located in source file /sys/sys/fio.c, decrements the reference count to the file. If there are no

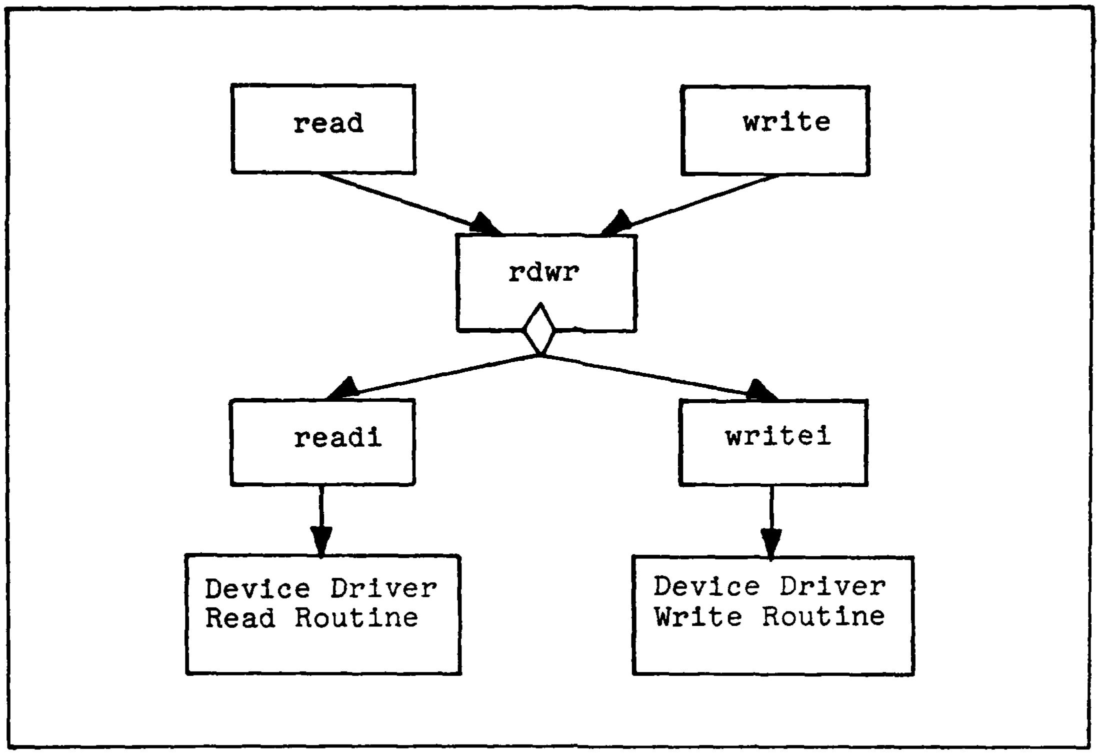


Fig 10. Routines for Processing the read(2) and write(2) System Calls

more references to the file then the system open file table entry is eliminated and the active i-node table entry is copied back to the i-list stored on disk. This is accomplished via a call to the "iput" routine (Ref 10:18-4). Finally, the closef routine invokes the device driver close routine via the appropriate device switch table. The minor device number is passed as an argument.

Read and Write. The read and write system call handlers are discussed together because they execute some common code. Figure 10 illustrates the routines invoked to process the read(2) and write(2) system calls. The read and write routines, located in source file /sys/sys/sys2.c, simply call the "rdwr" routine, passing a flag to indicate which routine made the call (Ref 10:18-4).

The rdwr routine, located in source file /sys/sys/rdwr.c, first checks the special file's access permissions to see if the read or write system call is permitted on that file. This is accomplished by using the file descriptor input parameter to check the special file's access permissions stored in the special file's active i-node. Next, the rdwr routine loads u.u_base with the address of the user buffer which was specified as the second input parameter of the system call. Next, u.u_count is loaded with the number of bytes to be transfered, i.e., the third input parameter of the system call. Next, rdwr sets up the offset into the user buffer by loading u.u_offset with the offset value obtained from the special file's active i-node. Finally, rdwr switches out to either readi or writei. These two routines are located in source file /sys/sys/rdwri.c. For character oriented special files, readi and writei simply switch out to the appropriate device driver read or write routines via the character device switch table.

Stty and Gtty. Figure 11 illustrates the routines called to process a stty or a gtty system call. The stty and gtty routines, located in source file /sys/dev/tty.c, each alter the u.u_arg[ ] array then call the "ioctl" routine. The u.u_arg[ ] array is altered because the ioctl routine expects a flag in u.u_arg[1] indicating whether the stty routine or the gtty routine made the call. Both stty and gtty alter the

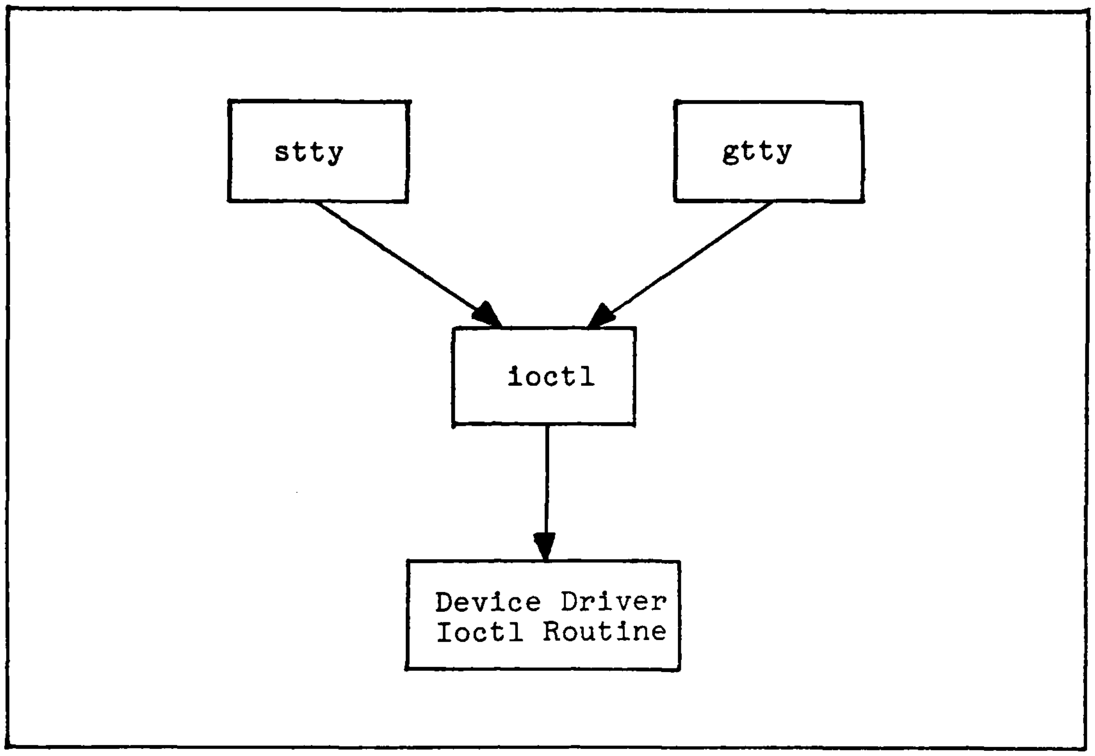


Fig 11. Routines for Processing the stty and gtty System Calls

u.u_arg[ ] array in the same way. The data in u.u_arg[1] is moved to u.u_arg[2], then the appropriate identification flag is placed in u.u_arg[1]. After this has been accomplished, the ioctl routine is invoked. This routine is located in source file /sys/dev/tty.c.

For character oriented special files, ioctl simply calls the appropriate device driver ioctl routine via the character device switch table, passing both the minor device number and the identification flag retrieved from u.u_arg[1]. The identification flag lets the device driver ioctl routine know whether the call is a stty or gtty call.

Device Switch Tables. The UNIX I/O handler routines call device driver routines via the system's device switch tables. Two such tables exist; the block device switch table (bdevsw) for block oriented devices and the character device switch table (cdevsw) for character oriented devices. In principle, the two tables are used in the same way. The cdevsw table is describe here.

The cdevsw table is declared in system source file /sys/h/conf.h and initialized in file /sys/conf/c.c. The following C code declares the table but does not dimension or initialize it.

```
1.  extern struct cdevsw {
2.                int (*d_open)();
3.                int (*d_close)();
4.                int (*d_read)();
5.                int (*d_write)();
6.                int (*d_ioctl)();
7.                int (*d_stop)();
8.                struct tty *d_ttys;
9.  } cdevsw [ ] ;
```

Lines 1-8 define a structure named cdevsw. The structure consists of seven elements (lines 2-8). Each of the first six elements is a pointer to a function that returns an integer value (Ref 7:114-116). The last element is a pointer to a tty structure.

Line 9 declares an undimensioned array of cdevsw structures (Ref 7:124). The array is named cdevsw, which may cause some confusion because each of the structures making up the array is also named cdevsw. Since the array is not dimensioned, no storage is allocated at this point.

The initialization of array cdevsw is defined in file

/sys/conf/c.c. The version of this file used when the VG graphics device is configured on the system, /sys/conf/c.c.vg, is included as Appendix C. The array is initialized to logically look like a table with 23 rows (0-22) of seven elements each (see lines 53-79, Appendix C). Each row in the table is reserved for a different character oriented peripheral device. The first six elements in each row are the names of the device driver routines for a particular device, while the seventh element is a pointer to a tty structure associated with that particular device.

The seven elements of the cdevsw structure map directly onto the seven elements of each row of the cdevsw table. In this way each row element may be referenced by specifying the corresponding name from the cdevsw structure. For example, the code

```
cdevsw[22].d_open
```

is a reference to the first element of row 22 in the character device switch table.

It has already been pointed out that the i-node for each special file contains a device class and device name. It has also been pointed out that the device class and major device number are used to determine which device driver routine to call. This concept is explained in detail here. The device class is either character or block. This indicates which switch table to use. The major device number is used as a row index into the appropriate table. For example, the i-node for a special file associated with the VG graphics device contains a device class "c" (for character oriented) and a major device number 22. This information tells the system that the names of the VG driver routines are found in row 22 of the cdevsw table. Row 22 contains the entries vgopen, vgclose, vgread, vgwrite, vgioctl, nulldev, and 0 (see line 77, Appendix C). Nulldev indicates that there is no driver routine for the d_stop function, while the zero entry indicates that no tty structure is needed for the VG graphics device.

The following C language statement is a general example of how the UNIX I/O handler routines call device driver routines via the cdevsw table.

```
(*cdevsw[maj].d_close)(dev);
```

In this example, assume "maj" contains the major device number and "dev" contains the minor device number obtained from a special file's active i-node. The statement evaluates to a function call on the device driver routine whose address resides in the d_close element of row maj in the cdevsw table. The minor device number in dev is passed as an argument.

The contents of the first set of parenthesis, *cdevsw [maj].d_close, evaluates to the address of a device driver routine The "*" is the C language indirection operator (Ref 7:89) and the "." is the structure member operator (Ref 7:120). Logically, *cdevsw[maj].d_close means get the value stored in the d_close element of row maj of the cdevsw table. For maj = 22, the code would return the address of the vg_close routine.

Once this address is fetched from cdevsw table, the vg_close routine is called with input parameter dev.

Summary. This section began with a very high level flow chart of how a user program I/O request is processed. Throughout this section the flow chart was expanded to show more detail. All of the routines discussed in this section are brought together in the form of a structure chart displayed in Figure 12. This chart represents all the main routines called to process user I/O system calls. Levels zero and one represent user level routines, levels two and three the trap handler routines, levels four through six the I/O system call handler routines, and level seven the generic device driver routines. The next section describes how peripheral device interrupts are processed by UNIX.

Processing Peripheral Device Interrupts

The high level flow chart for processing interrupts is expanded in Figure 13 to show more detail. This figure illustrates the types of routines called to process an interrupt and identifies the mechanism used to transfer control between each group of routines. An interrupt generated by the occurrence of an event on a periperal device causes a transfer of control to the UNIX interrupt handler routine via the device's interrupt vector. Control is transfered from the UNIX interrupt handler to the appropriate driver interrupt handler via the same interrupt vector.

The remainder of this section describes the peripheral

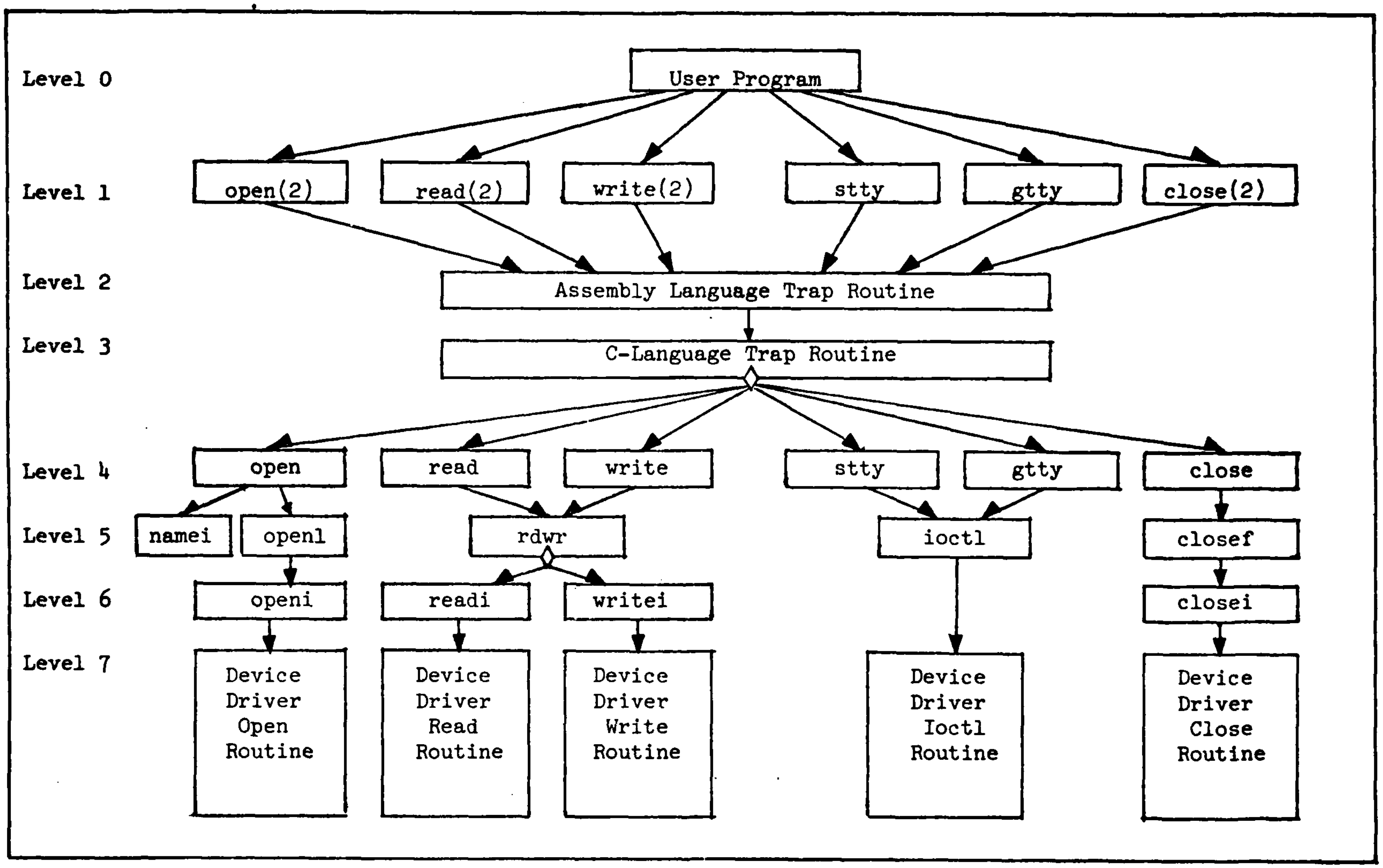


Fig 12. Main Routines Called During Peripheral I/O Processing

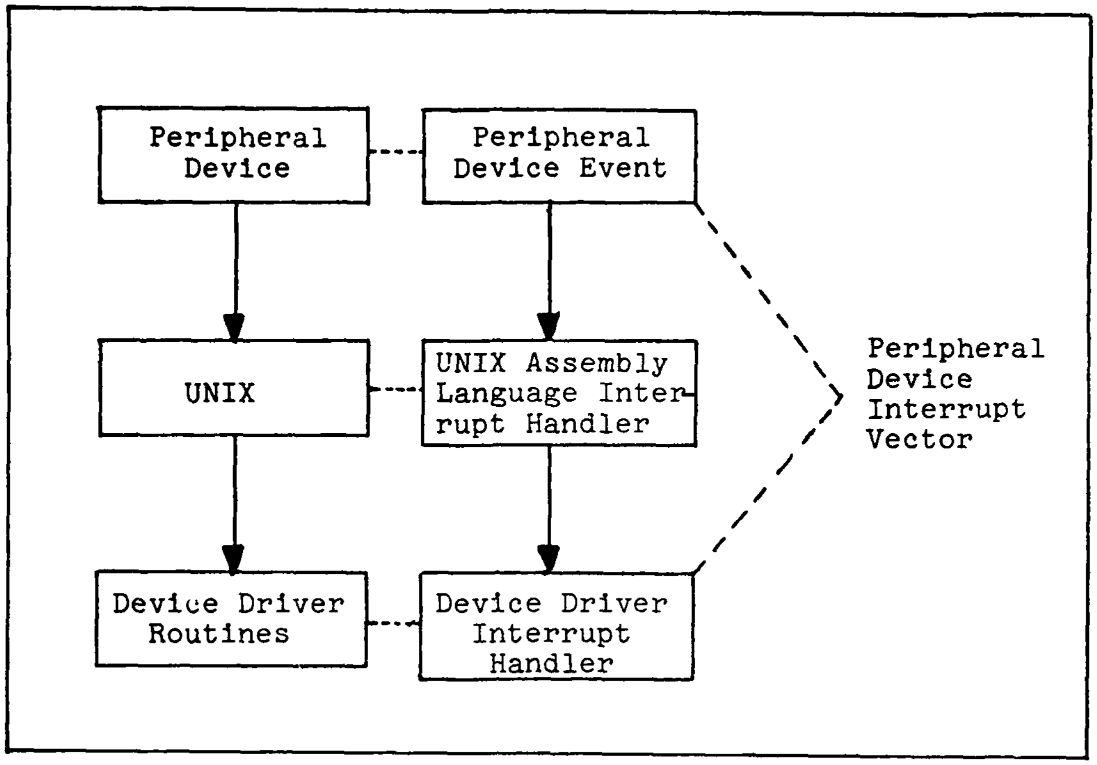

Peripheral Device
Peripheral Device Event
UNIX
UNIX Assembly Language Interrupt Handler
Peripheral Device Interrupt Vector
Device Driver Routines
Device Driver Interrupt Handler

Fig 13. Interrupt Processing Routines and Control Transfer Mechanism

device events, the device interrupt vector, and the UNIX interrupt handler routine. The VG's device driver interrupt handler, vgint, is described in detail in chapter six.

Peripheral Device Events. As opposed to system traps, interrupts result from events external to the CPU. External peripheral device events generate interrupts to get the attention of the CPU. The CPU is deverted from whatever it was doing and redirected to execute another program to process the event that caused the interrupt (Ref 10:9-1).

A number of different events occuring on a peripheral device may generate an interrupt. Some typical ones are input, output, device errors, etc. The types of events that generate interrupts are dependent on the type of peripheral device. Some peripheral devices may not have interrupt generating capability, while others may only support a few types of interrupts. Some devices support a wide range of interrupt generating events and even allow the user to "turn on" and "turn off" the interrupts for selected events (Ref 17:2-82 to 2-85).

The System Interrupt Mechanism. The system interrupt mechanism allows external devices to interrupt the CPU. Each device has an interrupt vector associated with it which is used to transfer control during interrupt processing.

Peripheral device interrupts are assigned a priority level 4, 5, 6, or 7. This priority is determined by the hardware (Ref 10:9-2). The processor also has a priority level associated with it from 0 to 7. This priority is carried in the current processor status (PS) word, bits 7 to 5 (Ref 10:9-2).

When a peripheral device generates an interrupt, the interrupt is inhibited as long as the processor priority is greater than or equal to the interrupt priority (Ref 10:9-2). When the processor priority becomes less than the interrupt priority, the interrupt is recognized. The processor then goes to the appropriate interrupt vector location to fetch new PS and PC values.

Different peripheral devices may have different interrupt vector locations. The location for a particular device is determined by hard wiring (Ref 10:9-2). The interrupt vectors on the PDP11/60 are located in low core and are defined in

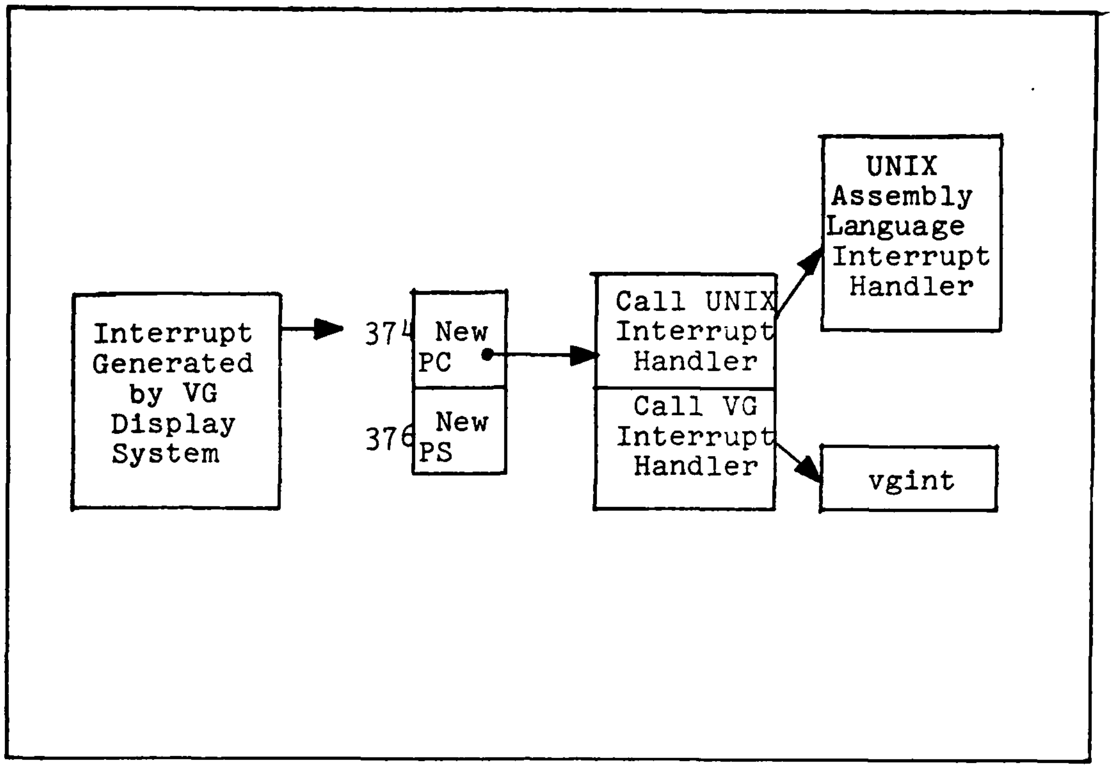

Interrupt Generated by VG Display System
374
New PC
376
New PS
Call UNIX Interrupt Handler
Call VG Interrupt Handler
UNIX Assembly Language Interrupt Handler
vgint

Fig 14. Interrupt Vector for VG Display System

the source file /sys/conf/l.s. A complete listing of the version of this file used for the VG graphics device, /sys/conf/l.s.vg, is given in Appendix B. For the VG graphics device, the new PC and PS values are loaded from octal locations 374 and 376 respectively (see lines 60-61, Appendix B).

The flow chart depicted in Figure 14 shows the transfer of control during interrupt processing. The VG graphic device's interrupt vector is used as an example. The processor loads new PC and PS values from the hard-wired vector location 374 and the word following that location, 376. After this step, the PC is pointing at a pair of interrupt handler calls (see line 84, Appendix B). The first is executed calling the UNIX device independent interrupt handler. When this routine is finished, the device dependent interrupt handler, vgint, which is part of the device driver software, is invoked.

The UNIX Interrupt Handler Routine. The UNIX interrupt handler consists of some of the same assembly language code executed to process system traps. This code is located in file /sys/conf/mch_i.s. For processing traps, the entry point to the code is label "trap". For processing interrupts, the entry point is label "call" (Ref 10:9-3). As with traps, the assembly language code first saves appropriate information on the system stack to be restored later. The device driver interrupt handler is then called to process the device dependent aspects of the interrupt.

Summary

This chapter described how the UNIX version seven operating system processes both user program requests for peripheral device I/O and peripheral device interrupts. This information is useful when developing driver software that runs under UNIX version seven.

Now that peripheral device I/O has been described, the next chapter discusses the VG 3404 peripheral device.

IV The Vector General 3404 Graphics Display System

Overall Description

The Vector General 3404 graphics device is a sophisticated graphics display system made up of the following major functional components (Ref 17:1-1).

1. Computer Interface
2. Graphic Processor Unit (GPU)
3. Refresh Buffer Unit (RBU)
4. Display Control Unit (DCU)
5. Vector Generator Unit (VGU)
6. Font Generator Unit (FGU)
7. Monitor Control Unit (MCU)
8. Display Monitor(s)
9. Display Input Device(s)
10. Options

Figure 15, derived from the Programming Concepts Manual (Ref 17:1-3), is a block diagram depicting the organization of the major functional components.

A user program builds a display list in the host computer's memory. This list is made up of instructions to be executed by the display system's Graphic Processor Unit (GPU). A complete list of the GPU instruction set is given in the System Reference Manual (Ref 18:3-7). The manual also provides a detailed description of each instruction (Ref 18:3-8 through 3-56).

After the display list has been built, the user program signals the GPU to start a one-time transfer of the display list from computer memory to the GPU via the computer interface. The GPU interprets the instructions in the display list and outputs a new list of elementary instructions called a

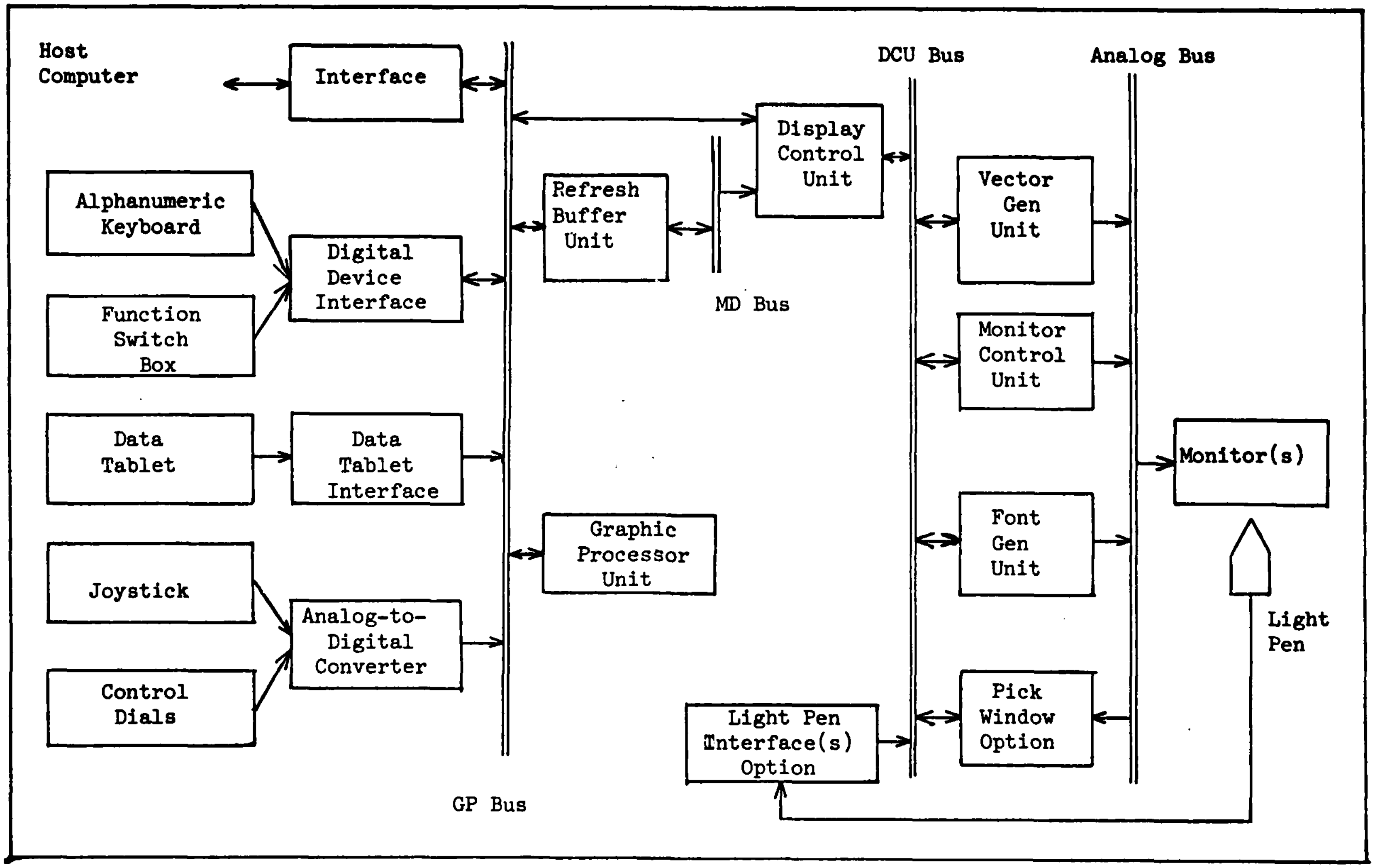


Fig 15. Display System Organization

refresh list. The refresh list is stored in the display system's Refresh Buffer Unit (RBU). The refresh list is repeatedly sent from the RBU to the Display Control Unit (DCU). The DCU interprets the refresh list instructions and causes the Vector Generator Unit (VGU) to draw lines, the Font Generator Unit (FGU) to draw characters, and the Monitor Control Unit (MCU) to control the CRT (Ref 17:1-1).

The remainder of this chapter is divided into two sections. The first section is a brief description of each of the display system's major functional components. This is followed by a discussion of the display system registers accessable for command, control, and communication purposes. Emphasis is placed on a description of the registers dealing with the display system input devices.

Functional Description of Major Display System Components

The display system's major functional components are described in both the Programming Concepts Manual and the System Reference Manual (Refs 17:1-2 to 1-7 and 18:2-2 to 2-5). Each major component is briefly described here.

Computer/Display System Interface. All communication between the host computer and the display system takes place via a hardware interface. The next chapter is devoted to a detailed description of the hardware interface between the PDP11 and the VG display system.

Graphic Processor Unit. The GPU is a high-speed special-purpose processor designed to handle complex algorithms such as transformations and other image manipulations. The GPU's instruction set consists of 47 basic instructions. User programs build display lists which consist of instructions from the GPU's instruction set along with any necessary data. The GPU fetches the user display list and associated data from the host computer's memory via a direct memory access (DMA) channel provided by the host computer/VG3404 hardware interface. The GPU processes the display list and outputs a refresh list to the RBU. Interaction between the GPU and RBU permits element selection and picking (Refs 17:1-2 and 18:2-2). All communication to and from the GPU takes place over the VG's Graphic Processor (GP) bus.

Refresh Buffer Unit. The RBU is made up of random access memory (RAM) and the control logic needed for reading and writing the RBU. The RBU may be continuously updated by the GPU over the GP bus.

The DCU accesses the RBU to update the displayed picture on the CRT screen. This takes place during each refresh cycle and does not interfere with the GPU updates to the RBU.

The RBU contains the necessary control logic to operate in double buffer mode. In double buffer mode, data may be moved from one buffer to the next when reorganizing the display refresh list for editing purposes.

Display Control Unit. The DCU fetches the refresh list from the RBU via the VG's MD bus. It processes the refresh list instructions and sends the appropriate refresh data to the VGU, FGU, and MCU. It also generates the control signals that cause the VGU, FGU, and MCU to display the refresh data. All of the communication between the DCU and the VGU, FGU, and MCU takes place over the DCU bus.

<u>Vector Generator Unit</u>. As stated in the Programming Concepts manual, "the VGU is a high speed vector generator which provides the deflection signals required to draw a line from one point to another on the face of the CRT" (Ref 17:1-5). The VGU operates on the x-y coordinate data it receives from the DCU via the DCU bus. It has the capability of generating curved lines on the display using a smoothing technique. It also performs the spacing between character positions as the FGU displays text.

<u>Font Generator Unit</u>. The FGU receives character codes, scaling, font, and rotation parameters from the DCU via the DCU bus. The character codes used are from the set of 96 ASCII characters.

The FGU uses a programmed ROM in conjunction with "stroke" character draws to display the characters on the screen (Ref 18:2-4).

<u>Monitor Control Unit</u>. The MCU selects the desired CRT for display and provides the required unblanking and intensity signals for the monitor video channel.

<u>Display Monitors</u>. The VG 3404 will support up to six CRT monitors per MCU. Optionally, up to eight CRTs can be supported (REf 17:1-6). AFIT only has one monitor at present.

<u>Display System Input Devices</u>. At present, AFIT's VG display system does not support the joystick, control dials,

nor light pen input devices. Nor does it have any remote input devices. The basic local input devices supported on AFIT's system are the alphanumeric keyboard, function switch box, and data tablet. These input devices all generate interrupts to the host CPU when they require service.

Options. The options available on the VG 3404 are listed in the Programming Concepts Manual (Ref 17:1-7). They include such things as additional input devices, additional RBU/DCU sets, pick facility, color monitors, etc. Aside from the input devices already mentioned, AFIT's display system has no other options.

Display System Registers

The VG contains many registers that can be read and written by the device driver and user programs to control display processing and to pass data and status information back and forth between the host computer and the display system. Each of these registers has a unique address in the display system. Each register is associated with one of the VG's major functional units. The registers are divided into two categories; (1) GPU registers and (2) hardware and device registers. A complete list of GPU registers, along with a description of each, is given in the System Reference Manual (Ref 18:3-57 through 3-70). The hardware and device registers are listed and described in both the System Reference Manual and Volume one of the Technical Manual (Refs 18:5-1 through 5-20 and 20:2-1 through 2-23). The purpose here is not to describe all of these registers. Users can learn the use of each register by studying the manuals. At this point, it suffices to say that the UNIX operating system and device driver software provide the capability for user programs to read and write any of the display system registers via the stty and gtty system calls. This capability is described in detail in chapter six.

The device driver software accesses some VG registers without being requested to do so by a user program. In particular, the device driver interrupt handler accesses the registers associated with the VG's input devices during interrupt processing. These registers are described in the Programming Concepts Manual (Ref 17:2-82 to 2-85). Only the input devices on AFIT's system are described here.

Data Tablet Registers. Three registers are associated with the data tablet. They are illustrated in Figure 16a. The display system addresses for these three registers are 1600-1602 (Ref 18:Appendix C).

The first two registers, DTX and DTY, hold the X and Y stylus positions respectively. These values are stored in the form of signed twos complement integers in the leftmost ten bits of the registers. These values are updated constantly as the stylus is moved around the data tablet.

The third register, DTS, holds control and status information. The XOS and YOS bits indicate that the stylus is out of bounds on the data tablet surface in the X and/or Y directions respectively. The PNN bit indicates the stylus

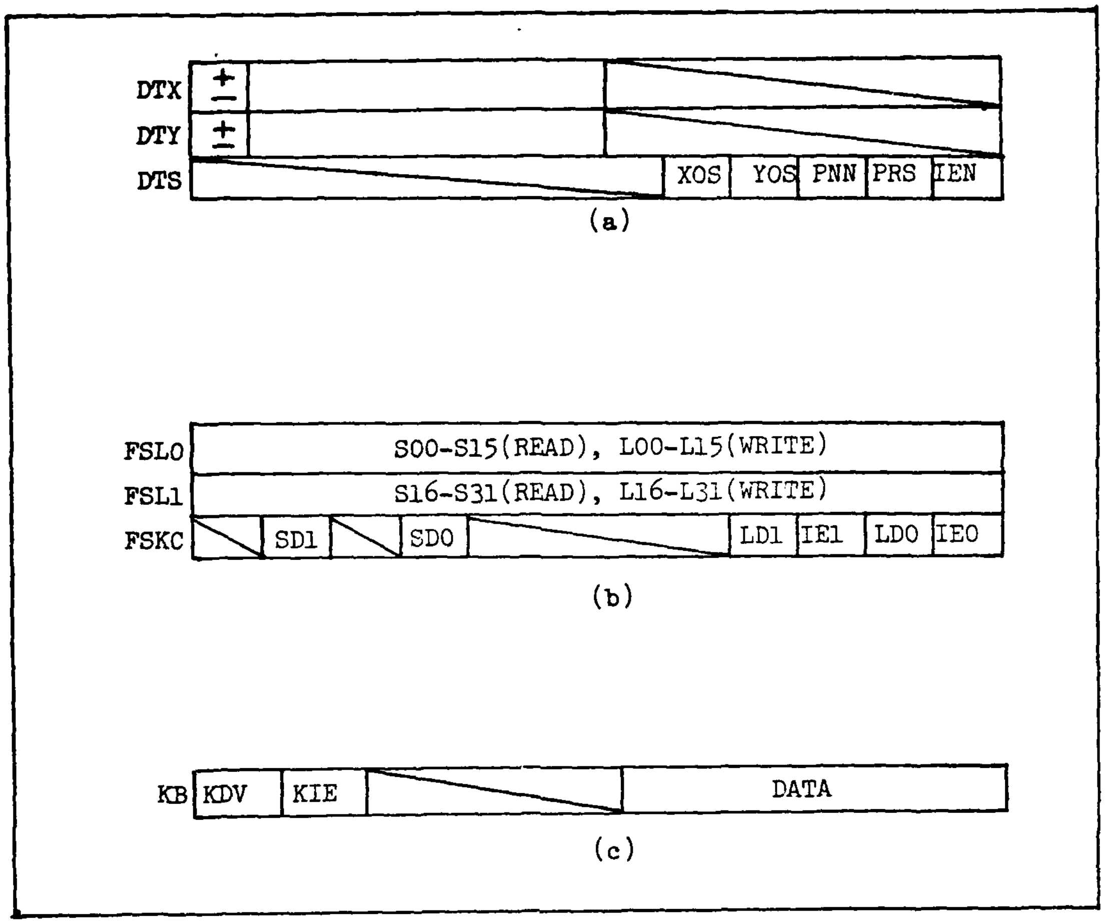


Fig 16. Input Device Registers

is within the "near" zone above the tablet. The PRS bit indicates the stylus switch is depressed. The IEN bit is set to enable interrupts generated by a change in the XOS, YOS, PNN, or PSS bits. It is set by the device driver program when a user program requests use of the VG data tablet.

<u>Function Switch Box Registers</u>. Three registers are associated with the function switch box. They are depicted in Figure 16b. Their addresses are 1604-1606 (Ref 18:Appendix C).

The sixteen bits of the first register (FSL0) correspond to the function switches S00-S15 and their respective lamps, L00-L15. The sixteen bits of the second register (FSL1) correspond to function switches S16-S31 and their respective lamps, L16-L31. The meaning of these two registers depends on whether they are being read or written by the device driver software. When reading, these two registers provide input data from the 32 function switches S00-S31. Every function switch depressed before the read causes the corresponding register bit to be set (Ref 17:2-83). When writing to these registers, all bits set to one turn the corresponding lamps (L00-L31) on.

The third register, FSKC, is for control and status. The IE0 and IE1 bits enable interrupts from the two switch groupings, S00-S15 and S16-S31 respectively. These bits are set by the device software when a program requests use of the function switches. SD0 and SD1 are sense bits which indicate a switch is latched in the S00-S15 and S16-S31 groups respectively. The LD0 and LD1 bits can be set to cause latching of all switches depressed in the S00-S15 and S16-S31 groups respectively. The latched data is cleared from FSL0 and FSL1 registers each time they are read by the device driver.

Alphanumeric Keyboard Register. One register is associated with the alphanumeric keyboard input device. It is illustrated in Figure 16c. The display system address for this register is 1607 (Ref 18:Appendix C).

The eight bit DATA field holds the ASCII code of the key depressed. The KIE bit enables interrupts for the keyboard. This bit is set by the device driver when a user program requests use of the keyboard. The KDV bit is set by the display system each time a key stroke has been latched in the data field. Reading the data field clears the KDV bit and allows another keystroke entry (Ref 17:2-84).

Summary

This chapter presented a functional description of the major components of the display system (except for the computer/VG3404 hardware interface component). A detailed description of the display system registers associated with AFIT's VG input devices was also given. The next chapter describes the PDP11/VG3404 hardware interface in detail.

V The PDP11/VG3404 Hardware Interface

Communication between the PDP11/60 computer and the Vector General 3404 graphics display system is established via the DE41 hardware interface (Ref 19). As stated in the DE41 reference manual, "this unit interfaces between the Unibus of any PDP11/60 computer and the General Purpose IO Bus of the NPL display controller" (Ref 19:3).

The interface provides four sixteen bit registers that can be directly addressed by the VG device driver running in the host computer. These are the Status, Control, Data, and Base Address registers. Using these four registers, the interface recognizes four input instructions and four output instructions. These eight interface I/O instructions, together with the four addressable interface registers, establish three channels of communication between the host computer and the VG display system. These three channels are the direct memory access channel (DMA), the interrupt channel (INT), and the programmed I/O channel (PIO). These three channels are illustrated in Figure 17 taken from the Programming Concepts Manual (Ref 17:2-2).

First, a detailed description of how to access the four interface registers from the device driver program in the host computer will be given. This is followed by an explanation of the use of the interface's eight I/O instructions. Finally, communication on the DMA, INT, and PIO channels is described.

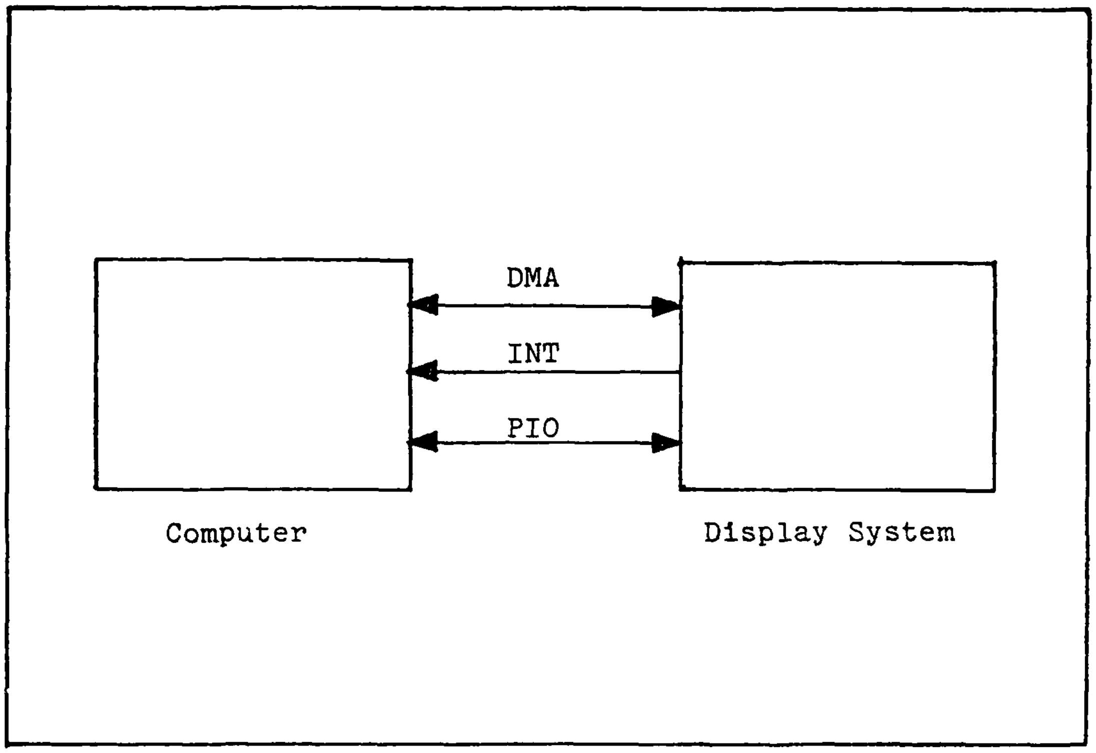


Fig 17. Interface Communication Channels

Accessing the Interface Registers

The interface's Status, Control, Data, and Base Address registers can be directly addressed by the VG device driver software executing in the host computer. These special registers are assigned physical addresses 0763400, 0763402, 0763404, and 0763406 from the highest page of the host computer's core memory. This is done because the highest page of core memory (addresses 0760000 to 0777777) is reserved for special registers associated with the processor and the peripheral devices (Ref 10:2-5).

Addresses from the highest page of the virtual address space (0160000 to 0177777) are mapped directly to the addresses of the highest page of the physical address space (Ref 10:2-5). Therefore, the interface's Status, Control, Data, and Base Address registers have virtual addresses 0163400, 0163402, 0163404, and 0163406 respectively. These virtual addresses are used in the device driver software to access the interface registers. The system takes care of mapping these sixteen bit virtual addresses to their eighteen bit physical addresses by adding in a base address obtained from the appropriate page register. Address mapping is described in detail in the section on the DMA channel.

The interface registers can be easily accessed from the device driver software. First, it is helpfull to associate meaningful names with the register addresses. This is accomplished in the C language with the "#define" macro substitution (Ref 7:86). The following C code was placed at the beginning of the device driver to associate names with the interface register addresses.

```
1.  #define VG_STAT 0163400
2.  #define VG_CONT 0163402
3.  #define VG_DATA 0163404
4.  #define VG_BAR  0163406
```

These statements tell the macro preprocessor, which is not part of the compiler proper, to replace all subsequent occurrences of the names VG_STAT, VG_CONT, VG_DATA, and VG_BAR with character strings 0163400, 0163402, 0163404, and 0163406 respectively.

VG_STAT, VG_CONT, VG_DATA, and VG_BAR are simply pointer values to the interface's Status, Control, Data and Base Address registers. In order to access the contents of the interface registers, the pointer values to the registers must be dereferenced (Ref 10:5-5). That is, the contents of the referenced location are desired instead of the reference itself. This is accomplished in the C programming language by creating a dummy structure consisting of one element, named "reg" (abbreviation for register), which is declared as type integer. The code

```
struct { int reg; };
```

is used in the device driver program (see line 99, Appendix D) to describe the dummy structure. This code does not cause storage to be allocated, it simply describes a template or the shape of a structure (Ref 17:120). A reference to the "reg" element of this template can be made using the structure pointer operator "->" (Ref 7:122).

The contents of the interface registers are accessed by specifying the appropriate pointer value (VG_STAT, VG_CONT, VG_DATA, or VG_BAR) connected to the "reg" element of the dummy structure by the "->" operator. To the C compiler, the code "VG_STAT->reg" means that 0163400 is the beginning address of an occurrence of the dummy structure. Since "reg" is the first and only element of the structure, its address is also 0163400. Therefore, the code "VG_STAT->reg" simply stands for the contents of address 0163400, i.e., the contents of the interface's Status register. The codes

"VG_STAT->reg", "VG_CONT->reg", "VG_DATA->reg", and "VG_BAR->reg" cause four separate occurrences of the dummy structure template to be overlayed on virtual memory at virtual addresses 0163400, 0163402, 0163404, and 0163406. The result of this code is represented pictorially in Figure 18.

The four interface registers are all read and written in the same way. For example, the C code statement

```
VG_DATA->reg = expression;
```

is used to load the interface's Data register. The word "expression" on the right side of the assignment operator can be a constant, variable name, function call, or any other legal C language expression. To read the same register the code

```
data = VG_DATA->reg;
```

is used; where "data" stands for some variable name.

The Interface's Eight I/O Instructions

Using the four addressable registers described above, the interface recognizes four input instructions and four output instructions (Ref 19:5). These are Status In and Status Out, Control In and Control Out, Programmed In and Programmed Out, and BAR In and BAR Out. The device driver executes these instructions by reading and writing the four addressable interface registers. All of the commands are executed over the interface's PIO channel. However, many of

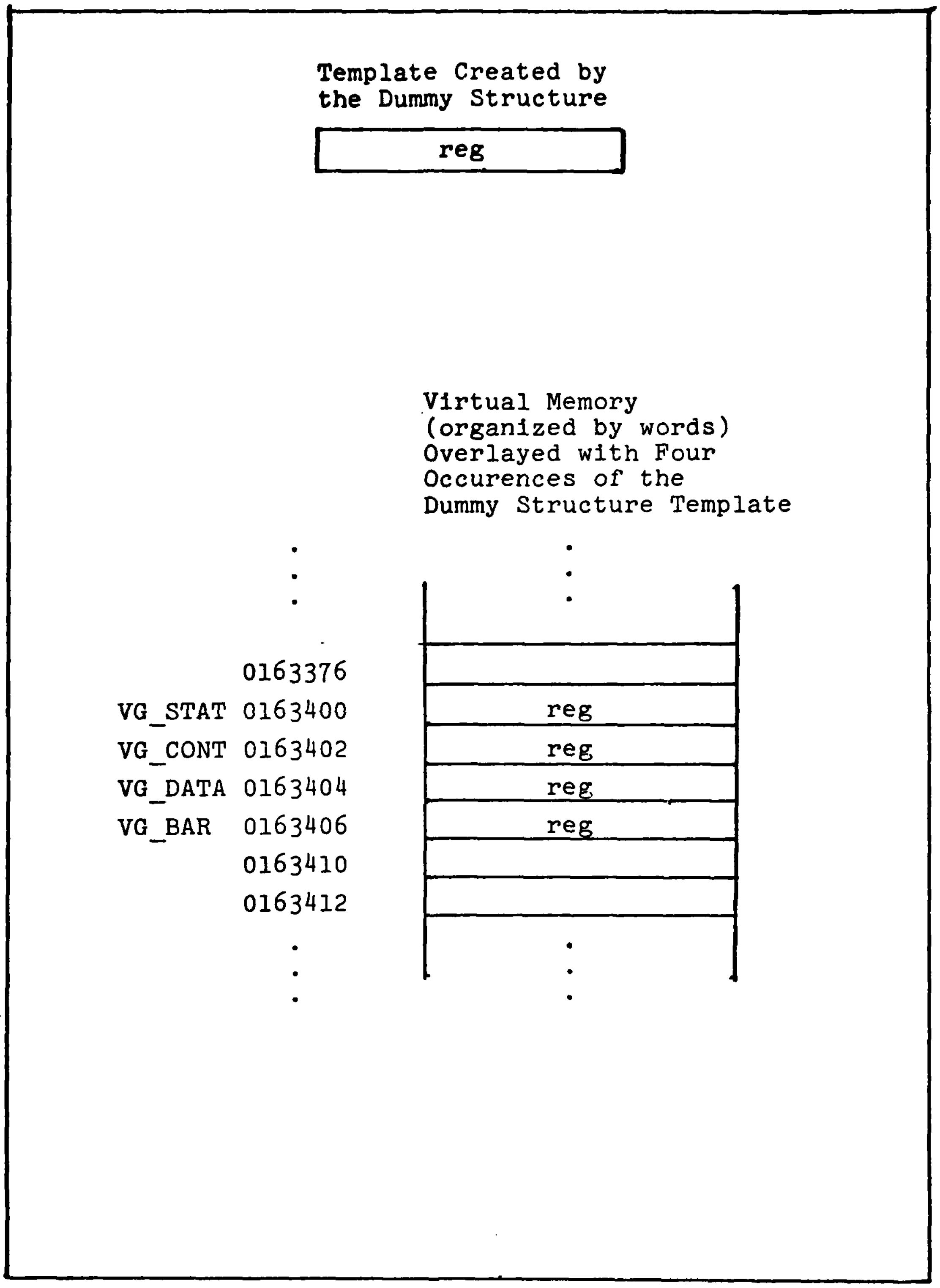

Template Created by the Dummy Structure
reg
Virtual Memory (organized by words) Overlayed with Four Occurences of the Dummy Structure Template
0163376
VG_STAT 0163400
reg
VG_CONT 0163402
reg
VG_DATA 0163404
reg
VG_BAR 0163406
reg
0163410
0163412

Fig 18. Using a Dummy Structure to Access the Contents of the Interface Registers

these instructions affect communication on the DMA and INT channels.

At this point it is worthwhile to mention that the four input instructions send input from the interface to the device driver, while the four output instructions send output to the interface from the device driver. In other words, the interface sends input to the device driver program and receives output from the device driver program.

A detailed description of the interface's eight I/O instructions can be found in the DE41 reference manual (Ref 19:5-7). A brief description of the purpose of each of the eight instructions is given here.

The Status In instruction (Ref 19:5) is used to obtain the ID of the last unit (within the VG display system) that interrupted the PDP11 processor. The Status Out instruction is used to restore the contents of the interface's Input Buffer Register (INR) after an interrupt has been processed.

Depending on which bits are set, the Control In and Control Out instructions (Ref 19:6) accomplish different tasks. Control In can be used to test whether the interface power is on, whether an input operation requested by the device driver program has been completed, or whether an output operation initiated by the device program has been completed. The Control Out instruction can be used to initialize the interface, enable new interrupt requests from the VG display system, acknowledge interrupts received from the VG display system, specify the address of a VG register so that it may be read or written, or request input from a VG register.

The Programmed In instruction (Ref 19:7) reads the contents of the interface's Input Buffer Register (INR). Programmed Out writes data to the VG register whose address was specified by the last Control Out instruction with the Register Change (RC) bit set.

The BAR In instruction (Ref 19:7) is used to read the interface's Base Address Register. BAR Out (Ref 19:7) is used to load the interface's Base Address Register. The function of the interface's Base Address Register is described in detail in the section on the DMA Channel.

Channel Communication

As mentioned earlier, the eight interface I/O instructions, together with the four addressable interface registers, are used to establish three channels of communication (DMA, INT, and PIO) between the host computer and the VG graphics device. The block diagram in Figure 19 illustrates which system components use each of the three channels. Communication may occur concurrently on all three of these channels (Ref 17:2-2). The purpose of this section is to describe what type of information flows over each channel, and how that flow of information is controlled.

The DMA Channel. The DMA channel is described in the Programming Concepts Manual (Ref 17:2-3). Primarily, the channel is used to pass the user defined display list from

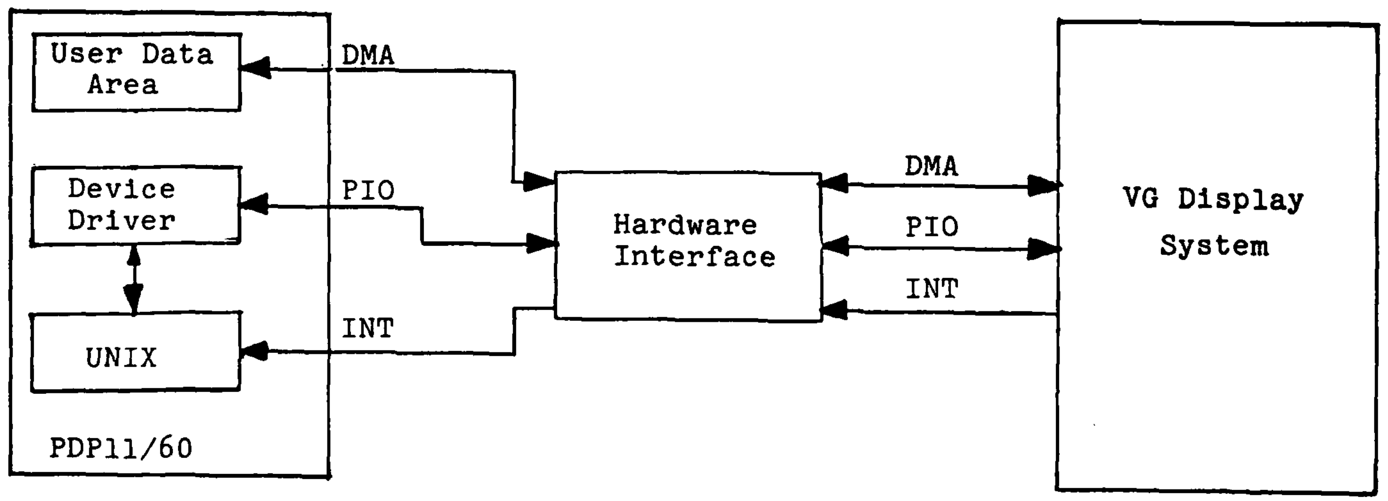


Fig 19. Communication Channel Usage

the host computer to the GPU in the display system. As the GPU processes the display list, it may fetch and/or store data in the host computer memory as required by the display list instructions. This data transfer also takes place over the DMA channel (Ref 17:2-3).

Memory addresses for DMA transfer are formed in the hardware interface by mapping 16 bit virtual addresses to 18 bit physical addresses. This is accomplished by adding the contents of the interface's Memory Address Register (MAR) to its Base Address Register (BAR). This address mapping is described in detail in the DE41 interface manual (Ref 19:13).

Before address mapping can take place, the BAR must be loaded with the proper base address. This address is obtained from a segmentation register in the host computer. The segmentation register used depends on whether the UNIX operating system has assigned the user program a sharable text segment or not.

If the user program has not been assigned a sharable text segment then the space allocated for the program to run is guaranteed to be mapped into contiguous memory and to begin at the zeroth page of the user's virtual address space. In this case the value loaded into the BAR is taken from the first User Instruction Space Address Register (UISA) located at virtual address 0177640 on the PDP11/60 (Ref 11:3).

If the user program has been assigned a sharable text segment, then the user space might not be mapped onto contiguous memory. In this case, the pointer to the user's text segment, u.u_procp->textp (see line 452, Appendix D), is used to calculate which segmentation register to use for loading the interface's BAR.

The VG device driver program checks for the two cases described above, then loads the BAR from the appropriate segmentation register with the BAR Out instruction. The code that accomplishes this task is discussed in the next chapter.

Once the BAR has been loaded, memory reads and writes can take place over the DMA channel. The following sequence of steps occur during a memory read (Ref 19:9).

1. The GPU requests use of the GP bus for a delayed data transfer.

2. Once the request is granted, the GPU sends a virtual memory address over the GP bus to the hardware interface's MAR.

3. The interface maps the 16 bit virtual address stored in the MAR to an 18 bit physical address. The base address stored in the interface's BAR is used during address mapping.

4. The hardware interface uses the 18 bit physical address to access PDP11 memory (via the UNIBUS) for the requested data. The retrieved data is placed in the interface's Input Buffer Register (INR).

5. Next, the interface requests the GP bus for a second data transfer. When the request is granted, the data is transferred from the interface's INR to the GPU.

Steps 1, 2, and 3 are the same for a memory write operation. Steps 4 and 5 are changed. The changes are listed below.

4. The interface reads the data from the requesting unit and places it in its INR. This is a separate GP bus transfer.

5. The interface uses the 18 bit physical address formed in step 3 to write the data from its INR to PDP11 memory (via the PDP11 UNIBUS).

Information flow on the DMA channel is controlled by interface I/O commands executed on the PIO channel, the content of the user defined display list, and the occurence of events within the display system. Commands sent over the PIO Channel may start and stop the transfer of the user defined display list from computer memory to the GPU. Instructions within the display list may alter the normal sequential processing from computer memory. The occurrence of a display system event, such as an interrupt from a display system input device, temporarily halts display list processing (Ref 17:2-3).

The Interrupt Channel. The Interrupt (INT) channel is described in the Programming Concepts Manual (Ref 17:2-3 to 2-4). The channel is used to signal interrupts to the PDP11 processor from the VG display system. A number of different events on the display system may generate interrupts. For example, keyboard, function switch, and data tablet inputs are all display system events that generate interrupts to the PDP11 processor. An interrupt is processed by the following steps (Ref 19:10-11).

1. A sub-unit of the VG display system signals an interrupt to the interface.

2. In accordance with priorities, the interface grants the GP bus to the requesting unit.

3. The requesting unit transfers a 6-bit interrupt identification to the interface's interrupt Identification Register (IDR). Subsequent interrupt requests are not honored by the interface until the current one is acknowledged by the device driver software.

4. The interface raises a priority level four interrupt request to the PDP11 Central Processor Unit (CPU).

5. In accordance with priorities, the CPU invokes the interrupt handler which is part of the device driver software.

6. First, the device driver interrupt handler, vgint, disables interrupts. Next it reads the interrupt ID from the interface's IDR to determine which VG sub-unit generated the interrupt.

7. As soon as the interrupt handler has acquired the interrupt ID from the IDR, it issues a Control Out instruction to set the interrupt acknowledge (ACK) bit in the interface's Control register. This acknowledges the interrupt and permits IDR to be changed by subsequent interrupts.

8. The interrupt handler performs the function required for the sub-unit that generated the interrupt then enables interrupts and returns.

The Programmed Input/Output Channel. The PIO channel is described in the Programming Concepts Manual (Ref 17:2-4 to 2-7). This channel is a bi-directional data path between the device driver software and the display system. The input path is from the display system to the device driver. The device driver controls the channel in both directions through the eight interface I/O instructions. The Programming Concepts Manual lists the following uses of the PIO channel (Ref 17:2-4).

1. Acquire status information.
2. Initialize the interface.
3. Read and write display system registers.
4. Start transfer of the user display list.
5. Control, categorize and acknowledge interrupts

The remainder of this section describes each of these capabilities.

Status information can be obtained about both the interface and the display system over the PIO channel. The interface's Control In instruction can be used to check for power on. Other status information about the display system is obtained by reading the appropriate display system registers.

The interface is initialized by executing a Control Out with the initialize (INIT) bit set (Ref 19:6). This is one of the first things done by the device driver.

Display system registers can be read and written over the PIO channel with the Programmed Input (PIN) and Programmed Output (POUT) routines. These routines are not to be confused with the Programmed In and Programmed Out instructions. PIN and POUT are invoked by the device driver to read and write display system registers, while Programmed In and Programmed Out are used by the device driver to read and write interface registers.

The PIN routine performs the following sequence of events (Ref 19:8)

1. Issue a Control Out setting the Control register's Request Input (RQI) bit equal to one, its Register Change (RC) bit equal to one, and its Register Number (RN) field equal to the address of a display system register. This causes the interface to

request data from the specified register. The data is transfered over the display system's GP bus. As long as the interface is still searching for the data, the Input In Process (IIP) bit is equal to one.

2. When the IIP bit equals zero, the requested data has been loaded into the interface's INR.

3. The data is read from INR with a Programmed In instruction.

The POUT routine consists of the following sequence of events.

1. Issue a Control Out setting the Control register's RC bit and loading its RN field with the address of the desired display system register.

2. The output data is loaded into the interface's Data register with a Programmed Out instruction.

3. Wait for the output process to complete by sensing the Output In Progress (OIP) bit of the interface's Control register. When it changes to zero the output process has been completed.

The PIO channel is used to start transfer of the user display list. First, a Programmed Out instruction is executed to load the interface's Base Address Register. Next, the POUT routine is used to load the GPU's Directory (DIR) and Picture Base Object (PBO) registers. Finally, the POUT routine is used to load the GPU's Command (CMD) register with the commands that cause the GPU to fetch the display list from computer memory (Ref 18:4-3).

A very important function of the PIO channel is the control, categorization and acknowledgement of interrupts (Ref 17:2-5). General interrupt handling can be enabled and disabled with the Control Out instruction. The POUT routine can be used to enable and disable particular types of interrupts by writing the appropriate value to the appropriate display system register. Interrupts are categorized by first obtaining the interrupt ID over the PIO channel using the Status In instruction. Interrupts are acknowledged over the PIO channel by invoking the Control Out instruction to set the Interrupt Acknowledge (ACK) bit in the interface's Control register.

Summary

This chapter described the hardware interface's four addressable registers, eight I/O instructions, and three communication channels.

Now that peripheral device I/O, the VG display system, and the PDP11/VG3404 hardware interface have been described, the device driver routines can be explained. This is accomplished in the next chapter.

VI The VG Device Driver

This chapter specifies the requirements for the VG device driver, describes the overall driver design, and documents the implementation of the driver.

Requirements

The VG device driver obtained from the University of Texas at Austin met certain requirements. This section identifies the original requirements then specifies the requirements adopted for AFIT's version of the device driver.

Original Requirements. The VG device driver obtained from the University of Texas was required to support two different levels of graphics. These two levels are depicted in Figure 20 (Ref 12:31).

With the level two graphics, the user display list consists of powerful GPU instructions. The GPU takes the instructions from the user display list and transforms them into a set of more fundamental instructions to be used by the DCU for display generation. The GPU performs the required two and three dimensional rotation, translation, windowing, clipping, curve generation, scaling, and sub-object definition management.

With level one graphics, the GPU is bypassed and the user display list is written directly into the RBU. This means that the user display list must consist only of the fundamental instructions understood by the DCU. This implies that the user program is responsible for performing all trans-

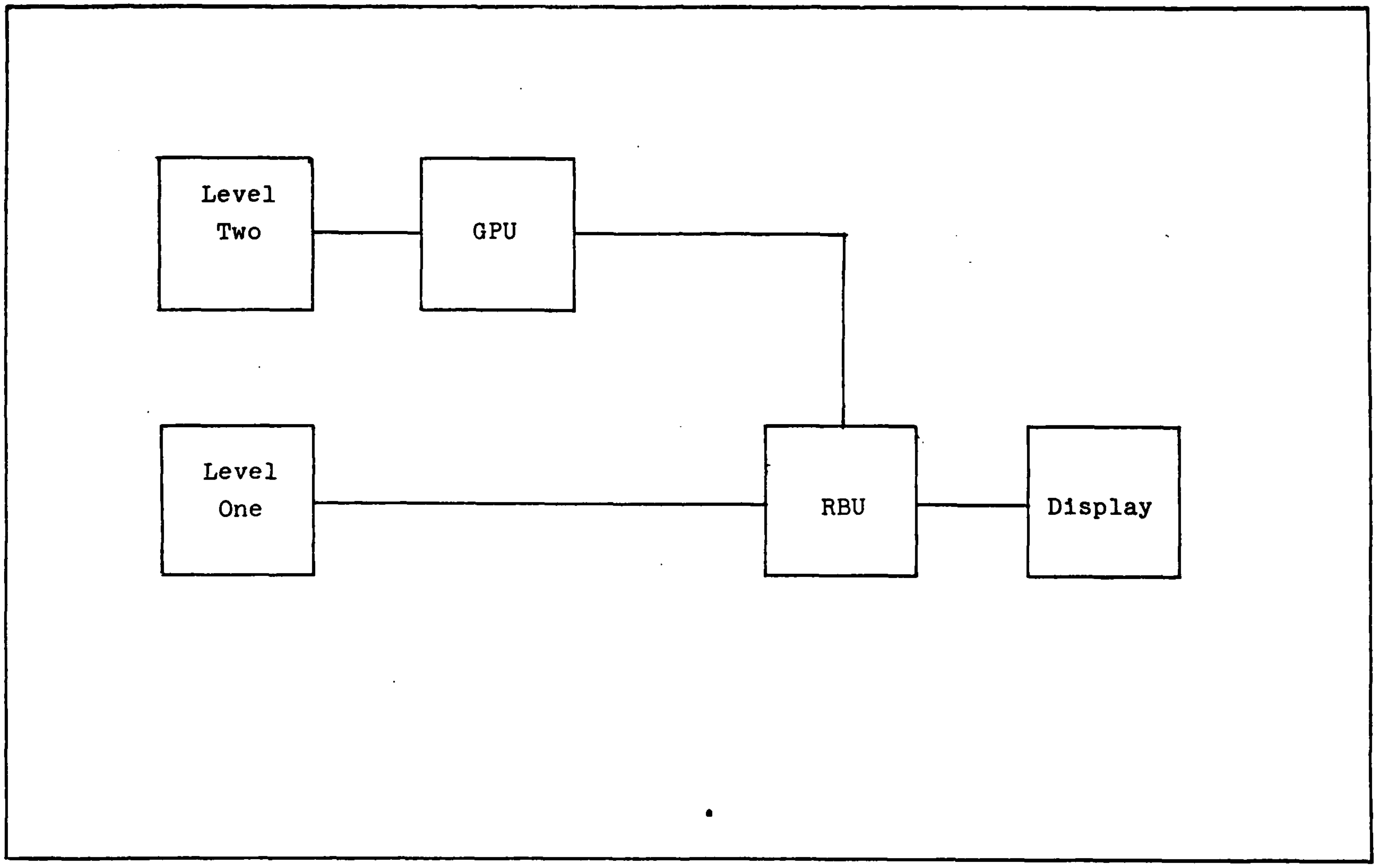


Fig 20. McCallum's Two Levels of Graphics Support

formations before building the display list. In order to support the level one graphics, a special DCU/RBU driver was installed. This driver provided the capability of reading and writing the RBU directly.

Another main requirement for the original driver was compatibility with the UNIX version six operating system. This included utilization of the standard UNIX interface for character oriented devices and support of the standard UNIX I/O system calls; open(2), close(2), read(2), write(2), stty, and gtty.

The original driver was also required to support the input devices available on the VG display system at the University of Texas. These consisted of a function switch box, an alphanumeric keyboard, and a light pen. Even though the original driver only incorporated the routines required for handling the available input devices, it was designed so that other input device handlers could be easily added.

The original driver provided most user programs the capability of reading and writing any addressable display system register. This capability is very important because the user program has to be able to load command and control information into display system registers. The user program must also be able to fetch display system status information and other data from display system registers. In conjunction with reading and writing display system registers, the original driver also allowed user programs to set and get certain device characteristics.

Requirements for AFIT'S VG Device Driver. Since a major objective of this thesis was to use as much of the original driver as possible, most of the original driver requirements were adopted for AFIT's version of the driver. Any changes that were made to the original requirements were mostly due to differences in the configuration of AFIT's system.

It was decided to not support the original requirement for a level one graphics capability. The main reason for this decision was that the original driver would not fit on AFIT's PDP11/60 due to limited space on the system. The level one graphics capability was selected for elimination because it did not use the display system's most powerful asset, the GPU. Elimination of the level one graphics did not degrade the system's capabilities, whereas elimination of the level two graphics would have limited the system's capabilities severely.

Another main requirement for AFIT's VG driver was compatibility with version seven of the UNIX operating system. To meet this requirement, some changes had to be made to the original driver. These changes are described in the next chapter.

AFIT's device driver was required to support the input devices available on AFIT's VG display system. These include the function switch box and alphanumeric keyboard supported by the original driver, plus the data tablet available on AFIT's system. Since the light pen was not available on AFIT's system, its interrupt handler was removed from the driver to conserve space. The ability to easily add new input devices to the system was maintained with AFIT's device driver.

The same UNIX I/O system calls that were supported by the original driver are also supported by AFIT's version of the driver. Although, the routines supporting the stty and gtty system calls had to be completely rewritten due to changes in the way UNIX version seven handles these calls.

Overall Design

The driver was designed in a top-down structured approach to facilitate programming and maintenance. It was designed for easy addition of more VG input devices such as the joystick and control dials.

The driver was designed around four sub-devices (also called minor devices) of AFIT's display system; the GPU, data tablet, alphanumeric keyboard, and function switch box. These minor devices were assigned minor device numbers 0, 1, 2, and 3 respectively. The structure chart in Figure 21 illustrates the routines needed to process user program I/O requests on the four minor devices and the routines needed to process interrupts generated by the four minor devices.

The UNIX routines represented by level zero of the structurechart were described in chapter three. The routines in level one of the structure chart are the major device routines called by UNIX. The major device routines call the minor device routines in level two of the structure chart.

The open(2), close(2), read(2), and write(2) system calls cause UNIX to call major device routines vgopen,

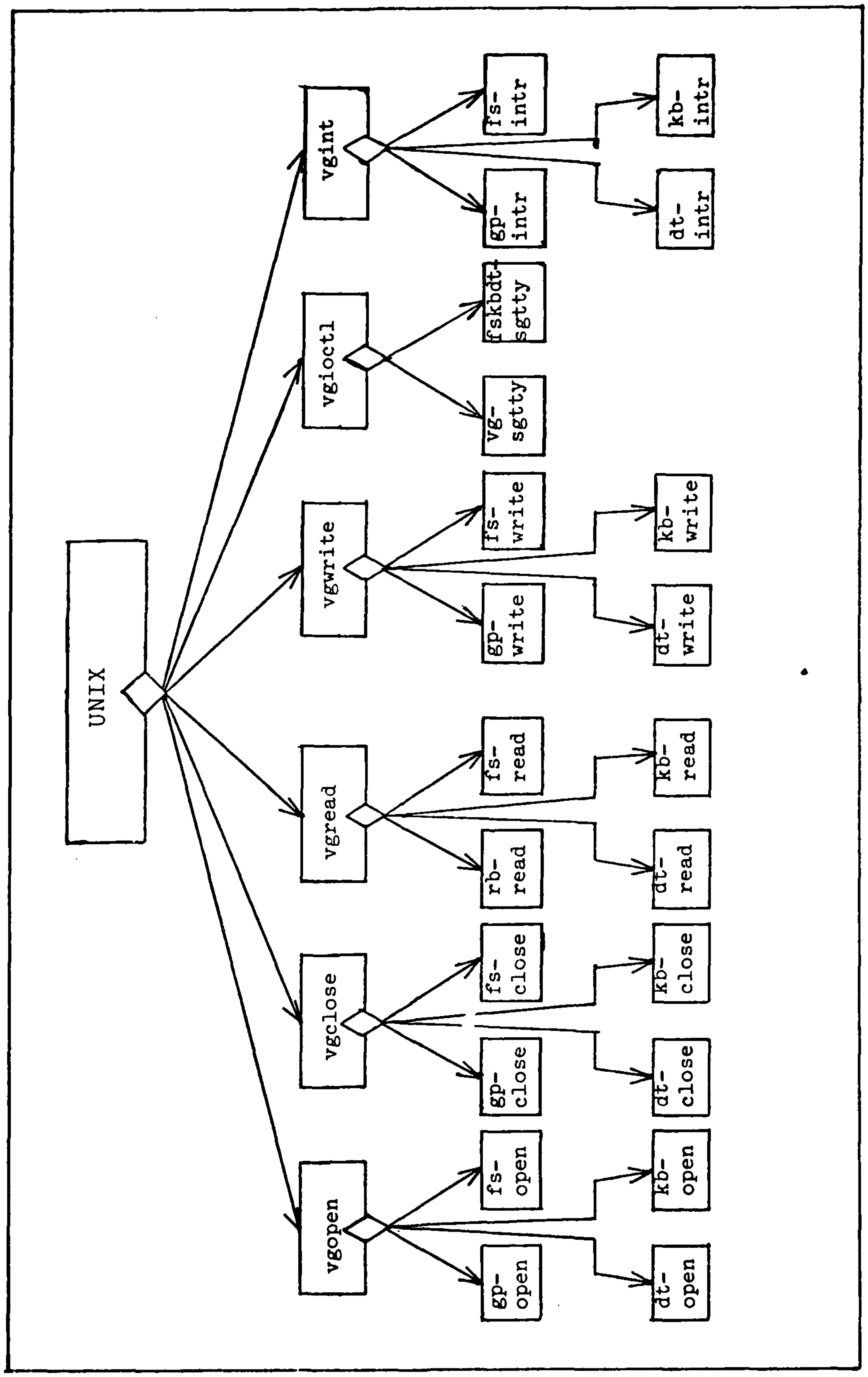


Fig 21. Device Driver Design

vgclose, vgread, and vgwrite respectively. The stty and gtty system calls cause UNIX to invoke the vgioctl major device routine. All of these major device routines are passed a minor device number which determines which minor device routines to call. For example, the I/O system call

```
open("/dev/gpu", mode);
```

causes UNIX to invoke the major device routine vgopen, passing it minor device number zero (for the GPU minor device). This minor device number causes vgopen to call minor device routine gpopen.

Display system interrupts cause UNIX to invoke the vgint routine. This routine determines which minor device generated the interrupt, then calls the appropriate minor device interrupt handler.

A new minor device can be added to the system by simply adding the new minor device routines to level two of the structure chart. For instance, if a joy stick input device is added to the sytem then minor device routines jsopen, jsclose, jsread, jswrite, and jsintr could be easily added to appropriate places in level two of the structure chart. A new character oriented special file, "/dev/jst", would be created with minor device number equal to 4.

With the overall design in mind, the implementation details are now presented.

Implementation

This section first describes the user level implementation details. This is followed by a complete description of the driver routines.

User Level Implementation. A character oriented special file was created for each of the VG minor devices (see Chapter VIII for details on creation of these special files). These files were named gpu, dtb, kbd, and fss for the GPU, data tablet, alphanumeric keyboard, and function switch box respectively. These four special files were created with major device number 22 which is the major device number associated with the VG display system. Minor device numbers 0, 1, 2, and 3 were assigned to the gpu, dtb, kbd, and fss special files respectively.

The four special files were all attached to the /dev directory. Therefore, they have pathnames /dev/gpu, /dev/dtb, /dev/kbd, and /dev/fss. A user program requests I/O on a VG minor device by first specifying the pathname of the associated special file as an input parameter to the open(2) system call. This opens the specified VG minor device for access. The open(2) system call returns a file descriptor to the user program to be passed as an input parameter on all subsequent I/O requests on the special file associated with the minor device. When finished with the minor device, the user program closes the associated special file by passing the file descriptor as an input parameter to the close(2) system call. The specific details of user program I/O on each of the four minor devices is presented next. All examples are given in the C programming language.

The GPU Minor Device. The GPU minor device is accessed via the special file /dev/gpu. User program I/O requests performed on this file are described here.

Open /dev/gpu. The /dev/gpu special file is opened for I/O access via a C language statement of the form

```
gpufd = open("/dev/gpu",2);
```

where gpufd (gpu file descriptor) represents a variable of type integer. The open(2) system call returns a file descriptor which is placed in variable gpufd. This file descriptor is used with all subsequent I/O requests on file /dev/gpu.

Read /dev/gpu. The /dev/gpu special file is read by a C language statement of the form below. The statement

```
m = read(gpufd,addr,n);
```

requests that n bytes be read from the VG display system's RBU and placed in a user buffer that has starting address addr. The number of bytes actually read is placed in integer variable m.

Write /dev/gpu. The write(2) system call is not supported on the GPU minor device. Therefore, an I/O error is flagged if a user program invokes the write(2) system call on special file /dev/gpu.

Stty /dev/gpu. When invoked on special file /dev/gpu, the stty system call is used to either write to a display system register addressed by the user or invoke one of five special functions. The form of the call is

```
stty(gpufd,info);
```

where gpufd is an integer variable containing the file descriptor that was returned when the GPU minor device was opened and info is the beginning address of an integer array of length three (int info[3]).

To write to a display system register, info[0] is loaded with the register address and info[1] is loaded with the data to be written. Next, the stty system call is invoked. The following statements illustrate how the call is invoked.

```
info[0] = display system register address;
info[1] = data;
stty(gpufd,info);
```

Five special functions may also be invoked with the stty system call. The following statements illustrate how a special function is invoked.

```
info[0] = function identifier;
info[1] = data (if required);
info[2] = data (if required);
```

Table I summarizes the five special functions available.

AD-A115 582 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
A UNIX BASED DEVICE DRIVER FOR THE VECTOR GENERAL 3404 GRAPHICS--ETC(U)
MAR 82 B R STEWART
UNCLASSIFIED AFIT/GCS/MA/81D-6 NL

2 OF 3
AD
A115582

Table I. Stty Special Functions

info[0]	info[1]	info[2]	Function Performed
Function Identifier	Data	Data	
-1	RBU Address	Value	Store value at RBU Address
-2	-	-	Perform RBU reset
-3	-	-	Suspend GPU processing of user's display list
-4	-	-	Restart GPU processing of user's display list
-5	Integer value from 0-15	-	Set the data tablet interrupt mask to the specified value

Gtty /dev/gpu. When invoked on special file /dev/gpu, the gtty system call is used to read display system registers. The following statements illustrate how a display system register is read via the gtty system call. In this example, info is a variable of type integer.

```
info = address of a display system register;
gtty(gpufd,info);
```

In the above example, the data read from the specified display system register is returned in the integer variable named info.

Close /dev/gpu. A user program closes the /dev/gpu special file with the following statement:

```
close(gpufd);
```

where integer variable gpufd contains the file descriptor returned when the file was opened.

The Data Tablet Minor Device. The data tablet minor device is accessed via the special file /dev/dtb. User program I/O requests performed on this file are described here.

Open /dev/dtb. The /dev/dtb special file is opened by a statement of the following form.

```
dtbfd = open("/dev/dtb",2);
```

This statement places a file descriptor in integer variable dtbfd (data tablet file descriptor).

Read /dev/dtb. The data tablet input device is read by a C-language statement of the following form.

```
n = read(dtbfd,&buf,m);
```

In this statement, m is the number of x-y coordinate pairs requested, buf is an integer array that must be at least 3m in length, dtbfd is an integer variable containing the data tablet file descriptor, and n is an integer variable which is assigned the actual number of x-y coordinate pairs read. For each x-y coordinate pair read, three pieces of data are returned; (1) an x coordinate value, (2) a y coordinate value, and (3) a data tablet interrupt ID which indicates which type of data tablet interrupt generated the x-y coordinate pair. During a read, these data "triples" are placed in the buf array. This is why the buf array must be at least 3m in length.

Table II. Data Tablet Interrupt IDs

Data Tablet Interrupt Type	Interrupt ID Returned To User Program
PRS	1
PNN	2
YOS	4
XOS	8

Four different types of interrupts may be generated by the data tablet input device; (1) the pressure switch on the data tablet stylus is depressed (PRS), (2) the data tablet stylus is within the "near" zone above the data tablet (PNN), (3) the data tablet stylus is moved off scale (i.e., out of bounds) in the y direction (YOS), or (4) the data tablet' stylus is moved off scale (i.e., out of bounds) in the x direction (XOS) (Ref 17:2-83). Table II contains the ID number for each type of data tablet interrupt.

A user program is allowed to specify which data tablet interrupts it will recognize. This is accomplished by invoking a special function via the stty system call. The following code illustrates how the special function is invoked.

```
info[0] = -5;
info[1] = data tablet interrupt mask value;
stty(gpufd,info);
```

This special function was already presented in the section entitled Stty /dev/gpu. Table III contains all the data

Table III. Data Tablet Interrupt Masks

Interrupt Mask Value	Interrupts Recognized			
	XOS	YOS	PNN	PRS
0				
1				X
2			X	
3			X	X
4		X		
5		X		X
6		X	X	
7		X	X	X
8	X			
9	X			X
10	X		X	
11	X		X	X
12	X	X		
13	X	X		X
14	X	X	X	
15 (default Value)	X	X	X	X

tablet interrupt mask values along with the respective data tablet interrupts recognized. Notice that with the default interrupt mask value (15) the user program recognizes all four types of data tablet interrupts.

The following C-language code is an example of a user program that reads one x-y coordinate pair from the data tablet input device. The x-y coordinate pair returned must be generated by a PRS interrupt from the data tablet stylus.

```
 1.  main( )
 2.  {int gpufd, dtbfd, n, buf[3];
 3.   gpufd = open("/dev/gpu",2);
 4.   dtbfd = open("/dev/dtb",2);
 5.   buf[0]=-5;
 6.   buf[1]=01;
 7.   stty(gpufd,buf);
 8.   n=o;
 9.   while (n<1) n=read(dtbfd,&buf,1);
10.
11.   close(dtbfd);
12.   close(gpufd);
13.  }
```

In this program the data tablet interrupt mask is set to 1 (lines 5-7). This ensures that only x-y coordinate pairs generated by a PRS interrupt will be returned to the user program.

A "while" control statement is used to execute the read(2) system call (line 9). This is done because the read(2) system call returns a -1 if no input data is available. The while statement continues to invoke the read(2) system call until input data is read. The while statement is necessary because the device driver software does not support a "time-out" on a read. That is, the device driver software does not wait for input if no input data is readily available when the read is invoked.

After data is read, buf[0] contains the x coordinate, buf[1] contains the y coordinate, and buf[2] contains the data tablet interrupt ID (which will be 1 in the example program above.)

Write /dev/dtb. The data tablet is a read only device. Therefore, an I/O error is flagged by UNIX if a user program attempts to write to the data tablet.

Stty /dev/dtb. The status of the data tablet input device can be changed by a user program via the stty system call. The form of the call is

```
stty(dtbfd, x);
```

where x is an integer variable containing a status value for the data tablet and dtbfd is an integer variable containing the file descriptor for file /dev/dtb.

Gtty /dev/dtb. A user program fetches the status of the data tablet input device via the gtty system call. The form of the call is given below.

```
gtty(dtbfd, x);
```

This call places the status of the data tablet in the integer variable x.

Close /dev/dtb. The data tablet minor device is closed by a user program with the following statement.

```
close(dtbfd);
```

The integer variable dtbfd contains the file description for file /dev/dtb.

The Alphanumeric Keyboard Minor Device. The alphanumeric keyboard input device is accessed via the special file /dev/kbd. User program I/O requests performed on this file are described here.

Open /dev/kbd. The /dev/kbd special file is opened by a statement of the following form

```
kbdfd = open("/dev/kbd",2);
```

This statement places a file descriptor in integer variable kbdfd (keyboard file descriptor).

Read /dev/kbd. The alphanumeric keyboard is read with a statement of the following form.

```
n = read(kbdfd,&buf,m);
```

In this statement, m is the number of characters requested, buf is an integer array of length m (into which the input characters will be read), kbdfd is an integer variable containing the file descriptor, and n is an integer variable which is assigned the actual number of characters read (or -1 if no input characters are available at the time of the read). The ASCII representation of the input character is the value returned to the user program.

Here again, a while statement may be used to wait for input to become available. For example, the C-language statements

```
n=0;
while (n<1) n=read(kbdfd,&buf,1);
```

continue invoking the read(2) system call until one character is read from the VG's alphanumeric keyboard input device.

Write /dev/kbd. The VG's alphanumeric keyboard is a read only device. If a user program attempts to write to it then UNIX flags an I/O error condition.

Stty and Gtty /dev/kbd. The stty and gtty system calls allow a user program to set and get the status of the alphanumeric keyboard input device. The forms of the calls are given below.

```
stty(kbdfd,x);
gtty(kbdfd,x);
```

These calls function just like the calls described under Stty /dev/dtb and Gtty /dev/dtb.

Close /dev/kbd. A user program closes the alphanumeric keyboard with the following system call.

```
close(kbdfd);
```

The Function Switch Box Minor Device. The function switch box input device is accessed via the special file /dev/fss. User program I/O requests performed on this file are described here.

Open /dev/fss. The /dev/fss special file is opened by a statement of the following form.

```
fssfd = open("/dev/fss",2);
```

This statement opens the function switch box input device and places a file descriptor in the integer variable fssfd (function switches file descriptor).

Read /dev/fss. The function switches are read with a statement of the following form.

```
n = read(fssfd,&buf,m);
```

In this statement, m is the number of values requested, buf is an integer array of length m (into which the input values will be read), fssfd is an integer variable containing the file descriptor, and n is an integer variable which is assigned the actual number of characters read (or -1 if no input values are available at the time of the read).

Once again, a while statement may be used to wait for input to become available. For example, the C-language statements

```
n=o;
while (n<1) n=read(fssfd,&buf,1);
```

continue invoking the read(2) system call until one value is read from the VG's function switch box input device.

Write /dev/fss. The VG's function switch box is a read only device. If a user program attempts to write to it, then UNIX flags an I/O error condition.

Stty and Gtty /dev/fss. The stty and gtty system calls allow a user program to set and get the status of the function switch box input device. The forms of the calls are given below.

```
stty(fssfd,x);
gtty(fssfd,x);
```

These calls function just like the calls described under Stty /dev/dtb and Gtty /dev/dtb.

Close /dev/fss. A user program closes the function switch box input device with the following call

```
close(fssfd);
```

In this statement, fssfd is an integer variable containing the file descriptor for file /dev/fss.

This ends the section on user level documentation. The next section documents the device driver routines.

The Device Driver Routines. A complete listing of the VG device driver routines is included as Appendix D. The description of these routines is divided into the following five sections.

1. Include Files
2. Global Data Structures
3. Common Procedures
4. Major Device Routines
5. Minor Device Routines

Include Files. Several "header" files containing global declarations are included as part of the device driver software. These files are included via the C programming language file inclusion operator, #include (Refs 7:86 and 10:1-3). The following eight header files are included in the device driver software (see lines 72-79, Appendix D).

1. param.h
2. buf.h
3. conf.h
4. dir.h
5. user.h
6. tty.h
7. proc.h
8. vg.h

These files are all located in the /sys/h directory. The first seven files contain global declarations for UNIX constants and structures, while the eighth file, vg.h, contains global declarations for display system constants. These files are referenced as needed throughout the remaining discussion of the device driver routines.

Global Data Structures. This section describes the global data structures used by the device driver software.

They are the UNIX proc and u structures, the vgunit array, and the VG minor device switch table (vgdev).

The UNIX proc and u Structures. The UNIX proc and u structures are used to pass data and control information back and forth between UNIX and the device driver software. The specific elements of these two structures referenced by the device driver software will be explained as they are encountered.

The vgunit Array. The vgunit array was created to keep track of activity on the four VG minor devices. This structure is defined below (also see lines 86-90, Appendix D).

```
1.  struct vgstruc {
2.      struct clist io;
3.      int status;
4.      int *vg_procp;
5.  } vgunit[4];
```

Lines 1-4 define a VG data structure, vgstruc, consisting of three elements; io, status, and vg_procp. Line five declares an array, named vgunit, consisting of four occurrences of the VG data structure; one for each of the four VG minor devices. Figure 22a illustrates the data structure created by this code. The minor device numbers are used as indices into the vgunit array. Therefore, vgunit[0] is associated with the GPU minor device, vgunit[1] with the data tablet minor device, etc. The purpose for the io, status, and vg_procp elements is now explained.

Each minor device has a first-in first-out (FIFO) queue associated with it for I/O purposes. The io element of each

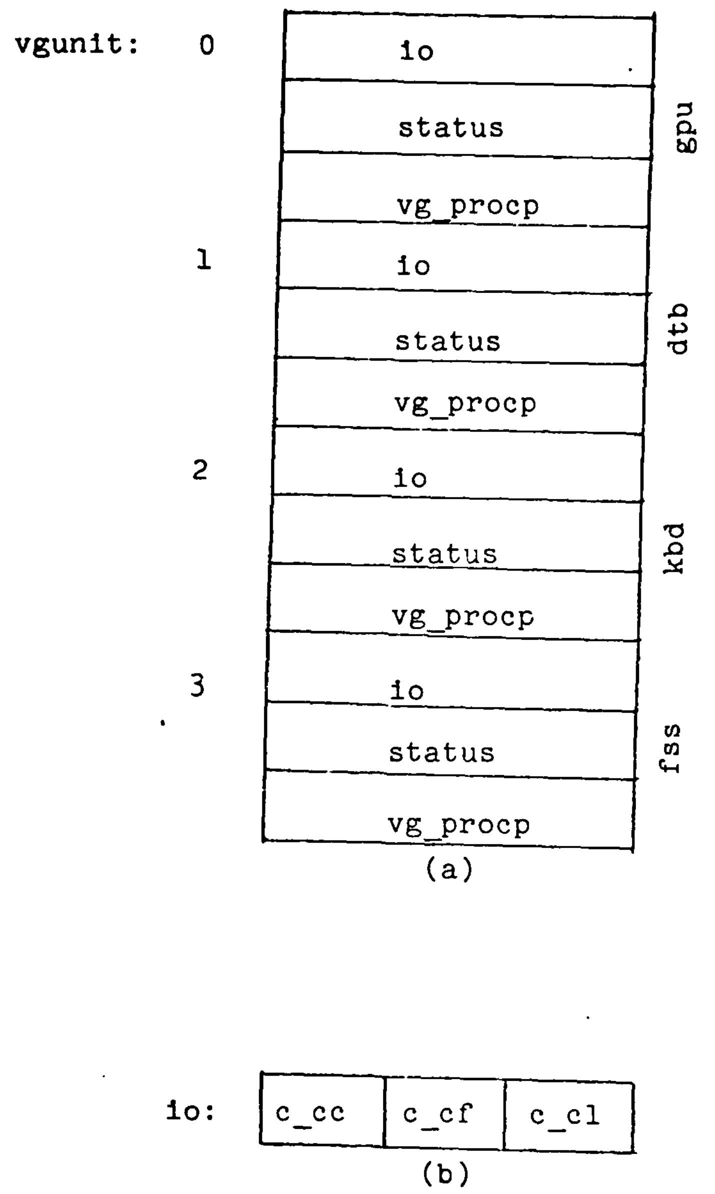


Fig 22. The vgunit Data Structure

minor device data structure is a header for the appropriate FIFO queue. For example, vgunit[1].io is a reference to the header of the data tablet's I/O queue while vgunit[2].io is a reference to the header of the alphanumeric keyboard's I/O queue.

Each io element is further broken down into three fields; c_cc, c_cf, and c_cl. The c_cc field contains the total number of elements in the FIFO queue, while the c_cf and c_cl fields contain pointers to the first and last elements of the FIFO queue respectively. Figure 22b illustrates the three fields of each io element.

The status element of each minor device data structure indicates whether the corresponding minor device is opened or closed. In the case of the GPU minor device it may also indicate whether the GPU is "running", "waiting", or "sleeping".

The vg_procp element of the minor device data structure is an indirect pointer to the proc structure of the user process that opened the minor device. It is an indirect pointer because it actually points at the u.u_procp element of the u structure which in turn points at the appropriate proc structure.

Use of the vgunit array will be explained more as the device driver routines are described.

<u>The VG Minor Device Switch Table</u>. The device driver software uses the UNIX idea of a device switch table for calling minor device routines. This table, named vgdev, is declared and initialized on lines 538-543 of Appendix D.

The table is declared as a cdevsw structure. This structure is defined in UNIX source file /sys/h/conf.h. The vgdev table is used exactly like UNIX's cdevsw table. That is, each row of the vgdev table contains the addresses of the open, close, read, write, and I/O control routines associated with a particular VG minor device. Row zero contains the routines for the GPU minor device, row one for the data tablet, row two for the alphanumeric keyboard, and row three for the function switches.

The minor device number passed to the major device routines by UNIX is used as an index into the vgdev table to select the appropriate set of minor device routines. The type of I/O system call determines which routine within the set is invoked.

Common Procedures. The following procedures are called from several different places in the driver software.

1. PIN
2. POUT
3. gpwait
4. gpurestart
5. putc
6. getc
7. passc
8. sleep
9. wakeup
10. psignal
11. fuiword
12. suiword

Routines 1-4 are defined in the device driver program while routines 5-12 are part of the UNIX source code. A brief description of each routine is given here.

PIN and POUT. The PIN and POUT procedures represent the implementation of the Programmed INput and Programmed OUTput functions described in the Programming Concepts Manual and the PDP11 Interface Specification (Refs 17:2-5 and 19:8).

The PIN procedure is used to read the contents of display system registers. The address of the register to be read is passed to PIN as an input parameter. First, PIN performs a Control Out instruction to load the register address into the Register Number (RN) field of the interface's Control Register. The Register Change (RC) bit, Request Input (RQI) bit, and Interrupt Enable (IE) bit of the interface's Control register are also set by the Control Out instruction. This causes the interface to request the desired data. PIN waits for completion of the input request then reads the data from the interface's Input Buffer Register (INR) with a Programmed In instruction. PIN returns this data to the routine that made the call.

The POUT procedure is used to write display system registers. The address of a display system register and the data to be written are passed to POUT as input parameters. First POUT performs a Control Out instruction to load the register address into the RN field of the interface's Control register and to set the RC and IE bits. Next, the data is written to the specified display system register via a Programmed Out instruction. Finally, POUT waits for the output operation to terminate then returns.

<u>gpwait and gpurestart</u>. The gpwait and

gpurestart procedures are used to stop and start GPU processing. The routines are called by the GPU, data tablet, alphanumeric keyboard, and function switch box interrupt handlers (gpint, dtint, kbintr, and fsintr).

The gpwait procedure is called to halt GPU processing temporarily. This ensures that the GP bus is free for processing an interrupt. The gpurestart procedure is called to restart the GPU processor.

putc and getc. The putc and getc routines are UNIX procedures written in PDP11 assembly language. The source code for these two routines is found in file /sys/conf/mch_i.s. The procedures are used to manage FIFO queues of 8-bit bytes.

The putc routine is used to add a character to a FIFO queue of characters. The procedure accepts two input arguments; (1) the address of a queue header, i.e., an io element within the vgunit array, and (2) the character to be added to the queue. Lion's describes in detail how the FIFO queue is set up and maintained (Ref 10:23-1 to 23-2). Here it suffices to say that putc takes care of allocating more space to the queue, adding the character to the queue, adjusting the queue pointers (c_cf and c_cl) stored in the queue header, and updating the queue count (c_cc).

The getc procedure is used to fetch characters from a FIFO queue of characters. The procedure is called with the address of a queue header as an input argument. The getc procedure takes care of all the overhead required to fetch a character from the specified queue. It fetches the next character from the queue, returns freed space to the available list, and adjusts the queue pointers and queue count stored in the queue header (Ref 10:23-2). If the queue is empty then getc returns a minus one, otherwise it returns the character fetched from the queue.

passc. The passc routine is a UNIX procedure which passes back a byte of information to the user program (Ref 11:65). The data is placed in the location referenced by the contents of u.u_base. The procedure updates u.u_base, u.u_count, and u.u_offset. If u.u_count goes to zero, signaling the last byte of the user's read, then the procedure returns a minus one. Otherwise, it returns a zero.

sleep and wakeup. The sleep and wakeup procedures are described in detail by Lions (Ref 10:8-3). They are UNIX routines used to suspend and reactivate user processes.

The sleep routine is used to suspend the process that is currently running. The procedure accepts two input parameters; (1) the reason for "sleeping" and (2) the priority with which the process will run upon being "awakened".

The wakeup procedure is invoked to reactivate a "sleeping" process. The procedure is passed the reason for sleeping as an input parameter. As stated by Lions, the procedure "simply searches the set of all processes, looking for any processes which are "sleeping" for a specified reason, and reactivates these individually" (Ref(10:8-3). The "awakened" processes enter the scheduling queue at the priority specified when the process was put to sleep (Ref 11;20).

psignal. The psignal procedure is a UNIX procedure which signals a software interrupt to the system. A detailed description of software interrupts is given by Lions (Ref 10:13-1 to 13-6).

The psignal procedure accepts two input parameters; (1) a pointer to a proc structure and (2) an interrupt signal. UNIX recognizes 15 different software interrupt signals. They are defined in UNIX source file /sys/h/param.h. Psignal stores the specified interrupt signal in the p_sig element of the specified proc structure. The system checks p_sig periodically to determine if a software interrupt signal is pending. If there is, then it is processed. Only one software interrupt can be pending for a process at any given time (Ref 10: 13-1).

fuiword and suiword. The fuiword and suiword procedures are UNIX procedures written in PDP11 assembly language. These procedures are used to fetch and store 16-bit data words in the user address space.

The fuiword procedure (Refs 10:10-1 and 11:8) is passed a user space virtual address as an input argument. The procedure fetches and returns the contents of the location addressed by the input argument. If an error occurs in this process, the procedure returns a minus one.

The suiword procedure (Ref 11:8) is passed a user space virtual address and a data word as input arguments. The procedure stores the 16-bit data word in the specified location of the user address space.

Major Device Routines. The generic or major device routines for the VG display system are vgopen, vgclose, vgread, vgwrite, vgioctl, and vgint. These are the routines invoked by UNIX to process the device dependent portion of display system I/O. Each major device routine performs the functions that are common to all of their subordinate minor device routines, then calls the appropriate minor device routine. The vgopen, vgclose, vgread, vgwrite, and vgioctl routines use the minor device number passed from UNIX to call the minor device routines via the minor device switch table, vgdev. The vgint interrupt handler uses a case statement keyed on the interrupt ID to call the appropriate minor device interrupt handler routine. Each of the major device routines are now described.

vgopen. The vgopen routine (lines 548-559, Appendix D) is called by UNIX to process the device dependent portion of an open(2) system call. The routine is passed a minor device number as an input argument. The minor device number is used as an index into the vgunit array to check the status of the corresponding minor device. If the minor device has already been opened then an I/O error code is placed in u.u_error and a return is made to UNIX. Otherwise, the proc structure pointer, vgunit[mdev].vg_procp, is initialized to reference the proc structure of the process opening the minor device. Next, the appropriate minor device open routine is called via the vgdev table. After the minor device routine returns, display system interrupts are enabled. This is accomplished by performing a Programmed Out to set the Interrupt Enable (IE) bit of the interface's Control Register. Finally, the status of the minor device is set to OPEN then the routine returns control to UNIX.

vgclose. The vgclose routine (lines 584-590, Appendix D) is called to process the device dependent portion of a close(2) system call. UNIX passes a minor device number as an input parameter. The routine uses the minor device number to call the appropriate minor device close routine via the vgdev table, enables display system interrupts, and sets the status of the appropriate minor device to zero indicating that the minor device is now closed.

vgread. The vgread routine (lines 562-568, Appendix D) is called to process the device dependent portion of a read(2) system call. The minor device number passed as an input parameter is used to call the appropriate minor device read routine via the vgdev table, then display system interrupts are enabled.

vgwrite. The vgwrite routine (lines 573-579, Appendix D) is invoked as the result of a write(2) system call. The minor device number passed as an input argument is used to call the appropriate minor device write routine via the vgdev table, then display system interrupts are enabled.

vgioctl. The vgioctl routine (lines 628-632, Appendix D) is invoked as the result of a stty or gtty system call (see ioctl(2)). UNIX passes a minor device number and a flag as input arguments. The minor device number is used to call the appropriate minor device I/O control routine via the vgdev table. The flag indicates whether the call is a stty or a gtty call. This flag is passed on to the minor device I/O control routine.

vgint. The vgint routine (lines 595-627, Appendix D) is the major device interrupt handler for the VG display system. It is called by UNIX when a display system event interrupts the PDP11 processor. First, the interrupt ID is obtained by performing a Programmed In on the interface's Status Register. Next, the processor priority is set to level seven, the highest possible priority, to prevent all other interrupts from interfering with processing of the current interrupt.

Next, a case statement, keyed on the interrupt ID, is used to call the appropriate minor device interrupt handler. A Programmed Out is performed to set the interrupt acknowledge (AKC) and Interrupt Enable (IE) bits of the interfaces Control Register. Finally, the processor priority level is set low and control is returned to UNIX.

Unrecognized interrupts are processed by the case statement's default condition. An error message is printed and the interface is reinitialized.

This concludes the description of the major device routines. The minor device routines are now described.

Minor Device Routines. The routines associated with the GPU, data tablet, alphanumeric keyboard, and function switch box minor devices are described in this section. The routines are presented by minor device.

GPU Routines. The GPU minor device routines handle the GPU portion of the VG display system. The routines include gpopen, gpclose, rbread, gpwrite, and vgsgtty.

gpopen. The gpopen routine (lines 441-460, Appendix D) is called by vgopen. The routine locks the user process in core, initializes the interface, and loads the interface's Base Address Register (BAR).

The user process is locked into core to prevent process swapping during display system access. This is accomplished by ORing the SSYS and SLOCK flags (defined in /sys/h/param.h) into the p_flag element of the process's proc structure (Ref 11:3).

The interface is initialized by issuing a Programmed Out instruction to set the Initialize (INIT) bit of the interface's Control Register.

The interface's BAR is loaded from the appropriate PDP11 user space segmentation register. If the user process does not share a text segment then the base address is taken from the first User Instruction Space Address Register (UISA) located at virtual address 0177640. This register is called APR in the driver software (line 33, Appendix D). It contains the address of the zeroth page of the user address space.

If the user process does share a text segment with another process then the zeroth page of the user address space may not be contiguous with the rest of the user process. In this case, the address of the first page of the contiguous portion of the user's address space is calculated and loaded into the interface's BAR.

gpclose. The gpclose routine (lines 487-492, Appendix D) is called by vgclose. The routine clears the GPU command register (display system address 07), stops the transfer of the refresh list from the RBU to the DCU, and unlocks the user process from core.

The GPU command register is cleared by writing zeroes to it via a Programmed Output (POUT). Transfer of the refresh list is inhibited by writing a 010 to the START/STOP field of the DCU control register (display system address 0400) (Ref 20:2-6). The user process is unlocked from core by removing the SSYS and SLOCK flags from the appropriate proc structure's p_flag element.

rbread. The rbread routine (lines 463-475, Appendix D) is called by vgread to process a read request on the GPU minor device. However, this routine really has nothing to do with the GPU. Instead, it allows a user program to read the contents of the RBU, i.e., read the refresh list. This capability was provided so that the refresh list could be read out, converted to raster form, and displayed on a raster scan device.

The routine uses u.u_count, u.u_base, and u.u_offset which contain the number of words to be read, the address of a user buffer, and current offset in the file. If u.u_count and u.u_base do not start on a word boundary then they are rounded down to the next word boundary. The routine reads u.u_count words from the RBU starting at u.u_offset. The data read is placed in the user buffer addressed by u.u_base. This is accomplished with the passc routine described earlier.

The POUT procedure is used to load the RBU's memory address register (rbumar) with the address from u.u_offset. This causes the contents of the addressed RBU word to be loaded into the RBU's data Register (rbudat). The PIN procedure is used to read the contents of rbudat. The data read is placed in the user's buffer with the passc routine. This entire process is repeated until u.u_count words have been read from the RBU.

<u>gpwrite</u>. The gpwrite routine (lines 479-482, Appendix D) is called by major device routine vgwrite. Since the GPU minor device cannot be written, the routine simply loads u.u_error with the I/O error flag, EIO, and returns.

<u>vgsgtty</u>. The vgsgtty routine (lines 100-147, Appendix D) is called by major device routine vgioctl to process the device dependent portion of the stty and gtty system calls. The stty and gtty system calls are handled differently by UNIX version seven than by UNIX version six. Therefore, vgsgtty had to be completely rewritten.

The structure chart in Figure 23 illustrates the functions carried out by the vgsgtty routine. First, vgsgtty

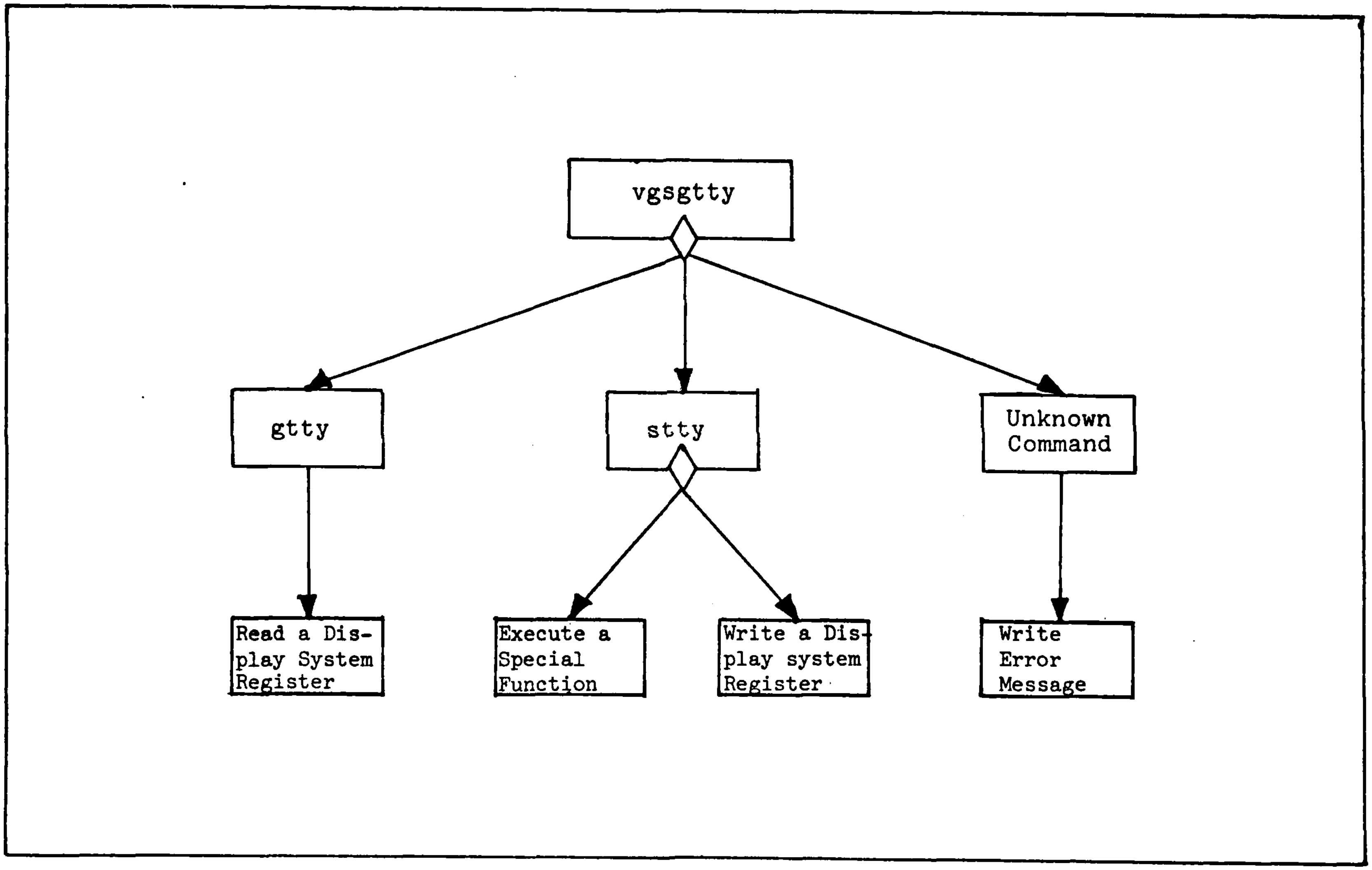

vgsgtty
gtty
stty
Unknown Command
Read a Display System Register
Execute a Special Function
Write a Display system Register
Write Error Message

Fig 23. Design of the vgsgtty Routine

retrieves the address of the user's data array from u.u_arg[2]. Next, the routine uses a case statement keyed on the flag input argument to control whether a gtty, stty, or unknown command is processed.

In the gtty case, the contents of a display system register are read and passed back to the user. This is accomplished by fetching the address of the display system register from the first location of the user's data array. This is done with the UNIX fuiword function. Next, the specified display system register is read with the PIN routine. Finally, the data is passed back to the first location of the user's data array via a call to the UNIX suiword function.

For the stty case, all three words of the user data array are fetched via three calls to the fuiword function. A case statement keyed on the value retrieved from the first location of the user's data array determines what to do next. If the case value is -1, -2, -3, -4, or -5, then the associated special function is executed; otherwise, a display system register is written.

With the -1 case, the data retrieved from the third word of the user's data array is written to the RBU location addressed by the value retrieved from the second word of the user's data array. This is accomplished with the POUT function.

For the -2 case, a call is made to the RBU reset procedure, RBURSET. This function resets the RBU by clearing both of the RBU's buffers. It also initializes the RBU's status and control registers (Ref 20:3-6).

The -3 and -4 cases invoke the gpwait and gpurestart routines respectively. These routines were described in the section on common procedures.

With the -5 case, the data retrieved from the second word of the user buffer is loaded into the data tablet's interrupt enable mask, dtintmask.

Any number of special functions can be added to the system by simply adding more cases to the software.

If the case value is not -1, -2, -3, -4, or -5 then it is interpreted as the address of a display system register. In this case, the data retrieved from the second location of the user's data array is written to the display system register addressed by the value retrieved from the first location of the user's data array.

Data Tablet Routines. The data tablet minor device routines handle the data tablet input device. They are dtopen, dtclose, dtread, dtwrite, fskbdtsgtty, and dtintr.

dtopen. The dtopen routine (lines 213-217, Appendix D) is called by vgopen to enable interrupts from the data tablet input device. This is accomplished by using a POUT to set the interrupt enable (IEN) bit of the data tablet's status (DTS) register (Ref 17:2-82).

dtclose. The dtclose routine (lines 221-225, Appendix D) is called by vgclose to disable interrupts from the data tablet and to flush the data tablet interrupt report queue. Interrupts are disabled by invoking the POUT function to clear the IEN bit of the DTS register. The interrupt report queue is emptied by invoking the getc routine on the queue until it is completely empty.

<u>dtread</u>. The dtread routine (lines 236-247, Appendix D) is called by vgread. The u.u_count variable contains the number of x-y coordinate pairs requested by the read(2) system call.

The dtread routine fetches an x-y coordinate pair and the associated interrupt identifier from the data tablet interrupt report queue and passes them back to the user buffer via calls to the UNIX passc procedure. This continues until u.u_count goes to zero or until the interrupt report queue is empty, whichever occurs first.

<u>dtwrite</u>. The dtwrite routine (lines 232-233, Appendix D) is called by vgwrite. Since the data tablet is a read only device, the routine simply loads u.u_error with the I/O error flag, EIO, and returns.

<u>fskbdtsgtty</u>. The fskbdtsgtty routine (lines 151-170, Appendix D) is called by vgioctl to process the device dependent portion of the stty and gtty system calls with respect to the function switch box, alphanumeric keyboard, and data tablet minor devices. As with the vgsgtty routine, the fskbdtsgtty routine had to be completely rewritten due to differences between UNIX versions six and seven.

The routine is passed a minor device number and a flag as input arguments. The flag is either TIOCGETP for a gtty call or TIOCSETP for a stty call. The minor device number is either 1, 2, or 3 for the data tablet, alphanumeric keyboard, or function switch box minor devices.

First, the routine retrieves the address of the user's data array that was specified in the system call. This address is retrieved from u.u_arg[2]. Next, a case statement keyed on the flag input argument is used to process either a stty or a gtty.

In the gtty case (flag=TIOCGETP), the status of the specified minor device is passed back to the first location of the user's data array via a call to suiword.

In the stty case (flag=TIOCSETP), the status of the specified minor device is set to the value retrieved from the first word of the user's data array.

dtintr. The dtintr routine (lines 250-264, Appendix D) is called by vgint to process interrupts generated by the data tablet input device. The data tablet's status register is read with a PIN to obtain the interrupt identifier. The interrupt identifier is ANDed with the data tablet interrupt mask, dtintmask, to see if the interrupt is recognized by the user program. If it is, then the data tablet's x and y data registers are read. The x-y coordinates and the interrupt identifier are then placed on the data tablet's interrupt report queue via calls to the UNIX putc routine. Before returning, the routine enables data tablet interrupts by setting the IEN bit of the data tablet's status register.

The Alphanumeric Keyboard Routines. The alphanumeric keyboard minor device routines handle the alphanumeric keyboard input device. They are kbopen, kbclose, kbread, kbwrite, fskbdtsgtty, and kbintr. The fskbdtsgtty routine was described under the data tablet routines. Therefore, it is not included here.

kbopen. The kbopen routine (lines 269-271, Appendix D) is called by vgopen to enable interrupts from the alphanumeric keyboard input device. This is accomplished by setting the KIE bit of the keyboard register (Ref 17:2-84).

kbclose. The kbclose routine (lines 276-280, Appendix D) is called by vgclose to disable interrupts from the alphanumeric keyboard and to flush the keyboard's interrupt report queue. Interrupts are disabled by clearing the KIE bit with a call to POUT. The interrupt report queue is emptied by repeatedly invoking the getc routine on the queue.

kbread. The kbread routine (lines 298-315, Appendix D) is called by vgread. The routine fetches a character from the keyboard's interrupt report queue and passes it to the user program via a call to the UNIX passc routine. This continues until u.u_count goes to zero or until the interrupt report queue is emptied, whichever occurs first. The routine will also terminate if the character read is a carriage return ('/n') or a control D ('/004').

kbwrite. The kbwrite routine (lines

294-297, Appendix D) is called by vgwrite. Since the alphanumeric keyboard is a read only device, the routine simply loads u.u_error with the I/O error flag, EIO, and returns.

<u>kbintr</u>. The kbintr routine (lines 320-326, Appendix D) is called by vgint to process interrupts generated by the alphanumeric keyboard input device. The routine uses the PIN function to read a character from the low order byte of the keyboard register. Next, the routine uses the putc function to place the character on the keyboard's interrupt report queue. Before returning, interrupts are enabled for the keyboard.

<u>The Function Switch Box Routines</u>. The minor device routines associated with the function switch box input device are fsopen, fsclose, fsread, fswrite, fsintr, and fskbdtsgtty. Once again, fskbdtsgtty has already been described under the data tablet routines.

<u>fsopen</u>. The fsopen routine (lines 337-340) is called by vgopen to enable interrupts from the function switch box input device. This is accomplished by setting the IE0 and IE1 bits of the Function Switch Control Register (FSKC) (Ref 20:2-22).

<u>fsclose</u>. The fsclose routine (lines 344-348) is called by vgclose to disable interrupts from the function switch box input device and to flush the function switch box interrupt report queue. Interrupts are disabled by using a POUT to clear the IE0 and IE1 bits of the FSKC register. The interrupt report queue is emptied by succes-

sive calls to the getc function.

<u>fsread</u>. The fsread routine (lines 364-392) is called by vgread to transfer u.u_count function switch readings to the user program. The routine first fetches a flag and a function switch value from the function switch box interrupt report queue. The function switch value is converted to an integer between 1 and 16. The flag indicates whether the function switch value came from the S00-S15 group or from the S16-S31 group. If the function switch value came from the S16-S31 group, then the value plus 16 is returned to the user program; otherwise, the value itself is returned. This continues until u.u._count goes to zero or the function switch box interrupt report queue is emptied, whichever occurs first.

<u>fswrite</u>. The fswrite routine (lines 359-362) is called by vgwrite. Since the function switch box is a read only device, the routine simply loads u.u_error with the I/O error flag, EIO, and returns.

<u>fsintr</u>. The fsintr routine (lines 397-412, Appendix D) is called by vgint to process interrupts generated by the function switch box input device. The routine determines whether the function switch depressed is in the S00-S15 pr the S16-S31 group. If in the S00-S15 group, the FSL0 register is read, else the FSL1 register is read. The contents of the register are placed on the function switch box interrupt report queue with a flag indicating which register the data came from. The routine enables

interrupts from the function switch box before returning.

Summary

This chapter presented the device driver requirements, device driver design, user level documentation, and documentation of the device driver code itself. The next chapter deals with the modifications made to McCallum's original software so that it would run on AFIT's system.

VII Device Driver Updates

Some changes were made to the original device driver obtained from the University of Texas so that it would run on AFIT's system. These changes fall into three categories; (1) changes due to space limitations on AFIT's PDP11/60, (2) changes due to display system differences, and (3) changes due to differences between UNIX versions six and seven. These three categories are discussed in this chapter.

Space Limitations

The original VG device driver would not fit under AFIT's current PDP11/60 system configuration. Therefore, the driver had to be trimmed downed in size. The following three features were removed from the original driver to make it smaller; (1) the level one graphics support, (2) the code intended to enable timeouts to occur during input device reads, and (3) the code supporting the VG light pen device.

Removal of Level One Graphics Support. In the search for ways to trim down the size of the device driver, it was decided to eliminate McCallum's level one graphics support. This level did not use the GPU and therefore did not exercise the full potential of the display system.

In the level one graphics support, the RBU was treated as minor device number 1 with rbopen, rbread, rbwrite, rbclose, rbsgtty, and rbintr as the minor device routines. The rbopen, rbclose, and rbwrite routines were removed from the driver software. Since the rbread and rbsgtty routines were shared with the GPU minor device, they were not removed. After removal of the RBU minor device, the name rbsgtty no longer had meaning. Therefore it was changed to vgsgtty.

The special function named staticopy was removed from the system. This routine was used to copy the static segment of the RBU from one RBU buffer to the other when the RBU was in double buffer mode. This was a rough attempt to allow static segments to run better for the level one graphics routines.

Removal of Code for Timeouts. The original driver contained code intended to allow a read invoked on a VG input device to timeout if no input was received within a specified period of time. The author of the original driver never got this feature to work. Therefore, it was removed to reduce the size of the driver software. The timeout code removed included the vgwait function, the tmoutdev function, the time and dtime elements from the vgunit structure, the constant GPU_timeout, and the code within the kbread and fsread routines intended for implementation of the timeout feature.

Removal of Code for the Light Pen. Since AFIT's VG display system does not have a light pen device, the code in the original driver supporting the light pen was removed to decrease the size of the driver. The following code was removed: the rbint routine which processed light pen interrupts; the cursup function and global variables vg_x, vg_y, vg_hitaddr, and vg_incr which were used to update the cursor after a light pen hit; and three special functions in the rbsgtty routine (subsequently changed to vgsgtty) for enabling light pen hits, disabling light pen hits, and loading vg_x and vg_y.

Display System Differences

AFIT's VG display system has a data tablet input device, but does not have a light pen device. The code added for the data tablet was described in the last chapter. The data tablet was assigned minor device number 1, which became available when the RBU minor device was removed from the system.

Differences Between UNIX Versions Six and Seven

Many differences exist between UNIX version six and UNIX version seven. These differences are transparent to users but not necessarily transparent to all device drivers. Several changes were required to make the original VG driver run under UNIX version seven. The differences between version six and version seven affecting the VG device driver are described here.

UNIX version six provided the following structure for accessing the low byte and high byte of a 16 bit computer word.

```
          struct{char lobyte; char hibyte;};
```

This structure was defined in the version six source file param.h. The version seven source file param.h does not contain the structure. Since the VG device driver references the structure often, it was added to the beginning of the driver software (see line 93, Appendix D).

UNIX version six also provided the structure

```
struct {char d_minor; char d_major;};
```

for accessing the major and minor numbers of a device name. This structure was declared in the version six source file conf.h. Version seven does not support this data structure. Instead it uses functions major(x) and minor(x) (declared in the version seven file param.h) for fetching the major and minor device numbers from the high order and low order bytes of the device name. In order to reduce the number of changes made to the original driver, the d_minor/d_major structure was added to the beginning of the driver software (see line 96, Appendix D).

Another difference between UNIX versions six and seven is the byte offset, u.u_offset. Thirty two bits are required to keep track of the byte offset in a file for I/O purposes. Since UNIX version six does not directly support "long" integers (i.e., 32 bit integer variables), it uses two sixteen bit words to keep track of the byte offset in a file. These two words are u.u_offset[0] and u.u_offset[1]. UNIX version seven does support long integers. Therefore, under version seven, u.u_offset is one variable declared as type long. As a result, all occurences of u.u_offset[0] and u.u_offset[1] in the original driver were changed to u.u_offset for compatability with UNIX version seven (see lines 469 and 473, Appendix D).

Two changes were made to the UNIX version six cdevsw structure; (1) a new element, d_stop, was added to the structure and (2) the d_sgtty element was changed to d_ioctl. These two changes affected the device driver software because the driver's vgdev table is declared as a cdevsw structure (see line 538, Appendix D). Since the d_stop element is not utilized by the driver a zero was placed in its position as a place holder in the vgdev table (see lines 539-542, Appendix D). The driver's only reference to the d_sgtty element of the vgdev table was changed to reference d_ioctl instead (see line 631, Appendix D).

Summary

Installation of a new version of UNIX which has an overlay capability is planned for AFIT's PDP11/60. This will relieve the limited space problem somewhat. At that time the level one graphics could be added back to the driver. The ability to support more systems software will allow for future expansion of the driver software to support other input devices such as the light pen, joy stick, and control dials.

The differences betwen UNIX versions six and seven did not cause any major changes to the original driver software. Nevertheless, a lot of research was required in order to understand the differences that affected the driver software.

Once the driver source code was updated, it was compiled and installed on the system. The next chapter describes the installation procedures in detail.

VIII Installing the VG Device Driver

The document entitled "Regenerating System Software" provides a general guideline for installing device drivers under the UNIX version seven operating system (Ref 4:6-9). The instructions in the document, together with a few modifications and additions, were used to install the VG device driver on AFIT's PDP11/60 computer.

Before the VG device driver could be installed, the system configuration had to be changed to make room. The existing system configuration at AFIT includes device drivers for the RK07 disk drives, the system console, and the time sharing terminals. This is the minimum configuration needed to support a multi-user, time shared environment. Unfortunately, this minimum configuration approaches the maximum size allowed for the UNIX operating system object file. The maximum allowable size is specified as 49,152 bytes (see line 57, Appendix E). In order to install the VG driver and remain within the size limit, something had to be removed from the current configuration. Since the RK07 disk drives and the system console are indespensable, the only thing that could be removed was the dh11 driver for the time sharing terminals. This means that in order to use the VG graphics device, the system must be degraded from multi-user to single user mode. This undesirable situation will be remedied in the near future with the installation of a new version of UNIX that provides an "overlay" capability. This capability will allow a larger

UNIX object file which, among other things, will support more device drivers.

The VG device driver was installed under UNIX version seven by performing the following eight steps.

1. Create the special files.
2. Relocate the driver source files.
3. Produce and archive the driver object file.
4. Edit the character device switch table.
5. Edit the interrupt vector file.
6. Produce the UNIX object file /unix.vg.
7. Restore the changed UNIX files.
8. Reboot the system from /unix.vg.

The remainder of this chapter is devoted to a detailed description of these eight steps. Figure 24 includes all the files and commands used during driver installation and identifies their location in the root file system. Use of these files and commands will be explained as the eight installation steps are described. A description of what was done to remove the dh11 driver for the time sharing terminals will also be given.

Creating the Special Files

Character oriented special files for the four VG minor devices were created via the system command mknod(1). The following system commands were executed to create the four special files.

```
1.  # cd /dev
2.  # /etc/mknod gpu c 22 0
3.  # /etc/mknod dtb c 22 1
4.  # /etc/mknod kbd c 22 2
5.  # /etc/mknod fss c 22 3
```

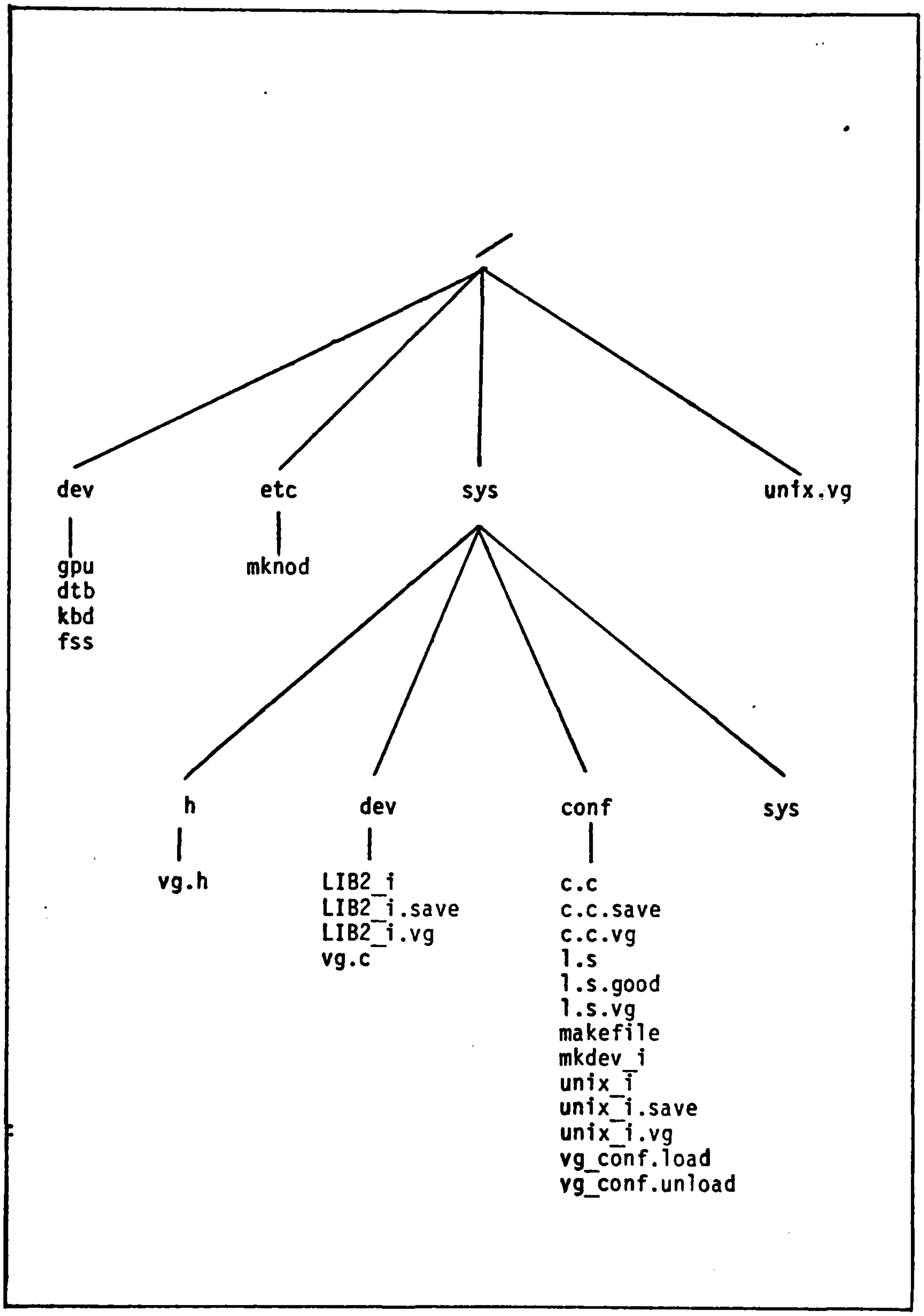


Fig 24. Location of Relevent Files and Commands

The system expects all character oriented special files to reside in the directory /dev. Therefore, the first command executed changed the current directory to /dev. Next, on lines 2-5, the four special files were created with the mknod command, which resides in the /etc directory.

The mknod command accepts four input parameters. The first parameter specifies the name of the special file to be created. The second parameter indicates that the file is character oriented as opposed to block oriented. The third and fourth parameters specify the file's major and minor device numbers respectively.

The mknod command uses the first input parameter to create a directory entry in /dev for the special file. Next, the mknod command creates an "inode" entry for the special file and stores it in the disk inode table (Ref 10:18-2). The file type (character in this case), major device number, and minor device number represent the file's characteristics. These characteristics are stored in the file's disk inode entry for later reference.

Appendix F contains a listing of directory /dev before execution of the mknod commands, a listing of the mknod commands, and a listing of /dev after execution of the mknod commands. The latter listing verifies the creation of the four special files.

The complete path names for the four special files are /dev/gpu, /dev/dtb, /dev/kbd, and /dev/fss. These complete pathnames are cited in user programs when referencing the VG graphics processing unit, data tablet, keyboard, and function switches respectively.

After creating the special files, their access modes were updated with the chmod(1) system command to allow all user programs to read and/or write them (Ref 5:13-14). The following list of commands were executed to accomplish this task.

```
1.  # chmod +rw /dev/gpu
2.  # chmod +r /dev/dtb
3.  # chmod +r /dev/kbd
4.  # chmod +r /dev/fss
5.  #
```

The special file associated with the VG graphics processing unit was given both read and write permissions, while the special files associated with the VG input devices were only given read permission.

Relocating the Driver Source Files

As stated by Haley and Ritchie, "the source and object programs for UNIX are kept in four subdirectories of /sys" (Ref 4:6). These four subdirectories are h, dev, conf, and sys. A complete listing of the contents of these subdirectories is given in Appendix G.

The subdirectory h contains header files which are picked up (via '#include ...') as required by each system module (Ref 4:7). These header files all end in '.h'. They contain global declarations needed by system modules (Ref 10:1-3). The VG device driver program obtained from the University of

Texas included a header file named initll.h. To maintain UNIX system standards, this file was renamed vg.h and was moved to subdirectory h. The resulting pathname for the file was /sys/h/vg.h.

The dev subdirectory consists mostly of device driver source files that all end in '.c'. The VG device driver source file, vg.c, was moved to the dev subdirectory. The resulting path name for the file was /sys/dev/vg.c.

Subdirectory conf is concerned with device configuration and will be described later in detail. Subdirectory sys contains the rest of the system and has nothing to do with device driver installation.

Producing and Archiving the Driver Object File

The directory /sys/dev contains two libraries, LIB2_i and LIB2_id, which contain all the device driver object files. LIB2_id is used with separate instruction and data (I and D) space CPUs while LIB2_i is used with non-separate I and D space CPUs, such as AFIT's PDP11/60. An object file for the VG driver was produced and archived in LIB2_i.

Before altering LIB2_i, an original copy of it was placed in LIB2_i.save. This guaranteed that LIB2_i could always be restored to its original state from LIB2_i.save.

In order to make room for the VG driver in the final UNIX object file, /unix.vg, the dh11 driver object file had to be removed from LIB2_i. This was accomplished with the archive system command, ar(1). The command line

```
# ar d LIB2_i dh.o
```

deleted the dh11 driver object file from library LIB2_i.

Next, a shell procedure (i.e. an executable file containing system commands) named mkdev_i was invoked to compile the VG driver source file and archive the resulting object file in LIB2_i. The following listing of mkdev_i reveals the commands executed by the UNIX shell program when mkdev_i is invoked.

```
 1.  # cat /sys/conf/mkdev_i
 2.  echo cp ../h/param_i.h ../h/param.h
 3.  cp ../h/param_i.h ../h/param.h
 4.  cd ../dev
 5.  touch junk.o
 6.  rm *.o
 7.  cc -c -o $1.c
 8.  cc -c -o $2.c
 9.  cc -c -o $3.c
10.  cc -c -o $4.c
11.  cc -c -o $5.c
12.  cc -c -o $6.c
13.  ar rv LIB2_i *.o
14.  rm *.o
15.  #
```

Line four changes the current directory to /sys/dev where all the driver source files reside. Lines seven through thirteen allow mkdev_i to compile and archive up to six device driver source modules at once (Ref 4:8). The '.c' extension required by the C compiler is automatically appended to the input files.

The following command stream was invoked to compile and archive the VG device driver in library LIB2_i.

```
1.  # cd /sys/conf
2.  # cp /sys/dev/vg.c /sys/dev/vg
3.  # mkdev_i vg
4.  cp ../h7param_i.h ../h/param.h
5.  a - vg.o
6.  # rm /sys/dev/vg
7.  # cp /sys/dev/LIB2_i /sys/dev/LIB2_i.vg
8.  #
```

The first command changed the current directory to /sys/conf where mkdev_i resides. On line two a copy of the VG driver source file was made in /sys/dev/vg. This was done because mkdev_i expects no '.c' extension on its input parameters. Mkdev_i was invoked on line three with file /sys/dev/vg as the input parameter. Line five is a message indicating that the VG driver object file, vg.o, was successfully added to LIB2_i. On line seven a copy of the updated LIB2_i was saved in LIB2_i.vg.

Editing the Character Device Switch Table

The system's character device switch table (cdevsw) is contained in the file c.c which resides in the device configuration directory /sys/conf. A complete listing of file c.c is given in Appendix C.

Each row in the cdevsw table is reserved for a particular character type device driver. The ordinal position of the row in the table implies the device's major device number, starting from 0 (Ref 4:9).

A row in the cdevsw table gives all the information the system needs to know about a particular device driver.

As stated by Haley and Ritchie,

> "For character devices, each line in the table specifies a routine for open, close, read, and write, and one which sets and returns device-specific status .... If there is no open or close routine, 'nulldev' may be given; if there is no read, write, or status routine, 'nodev' may be given. Nodev sets an error flag and returns." (Ref 4:9)

The system expects the name for the open routine to be in column one, the close routine in column two, the read routine in column three, the write routine in column four, and the status routine in column five.

Before altering the file /sys/conf/c.c, a copy of it was made in the file /sys/conf/c.c.save. This was done so that the original /sys/conf/c.c could always be restored to its original content from /sys/conf/c.c.save.

The names of the VG device handler routines (vgopen, vgclose, vgread, vgwrite, and vgioctl) were added as row 22 at the end of the existing cdevsw table (see line 77, Appendix C). Thus, the number 22 became the major device number for the VG graphics device. This explains why the number 22 was specified as the major device number when creating the special files associated with the four VG minor devices.

The code

```
int vgopen(), vgclose(), vgread(), vgwrite(), vgioctl();
```

was added to file /sys/conf/c.c to declare the VG driver routines to be of type integer (see line 51, Appendix C). Comments were added to the beginning of the file to indicate that the VG device handler had been added (see lines 1-11, Appendix C).

The routines for the dh11 driver were removed from the cdevsw table. These entries were replaced with 'nodev' and 'nulldev' as needed (see line 59, Appendix C). Also, the line declaring the type of the dh11 driver routines was deleted. A copy of the updated file /sys/conf/c.c was saved in /sys/conf/c.c.vg.

Editing the Interrupt Vector File

The file l.s, which resides in the /sys/conf directory, contains the system's device interrupt vectors. A complete listing of the file l.s is given in Appendix B. The interrupt vector for the VG device was added to this file.

Before altering the file l.s, a copy of it was made in the file /sys/conf/l.s.good. This guaranteed that l.s could be restored to its original state from l.s.good.

The interrupt vector for the VG device begins at location 374 (octal). When the VG interrupts the PDP11 CPU, the program counter (PC) is loaded with the value stored at location 374, while the processor status (PS) word is loaded with the value stored at location 376. The assembly language code

```
.= ZERO+374
        vgint; br7
```

was added to the file l.s to store the appropriate values at locations 374 and 376 (see lines 60-61, Appendix B).

The assembly language code

```
        .global _vgint
        vgint:  jsr     r0, call; jmp  _vgint
```

was added to file l.s to provide the capability of calling the VG device interrupt handler routine (lines 83-84, Appendix B).

The interrupt vectors for the dh11 and dm11 drivers were removed from the file l.s. The dm11 driver is used for modem devices and is directly coupled to the dh11 driver. After removing the dh11 driver, the dm11 interrupt vector was no longer needed. A copy of the updated file /sys/conf/l.s was saved in /sys/conf/l.s.vg.

Producing the UNIX Object File /unix.vg

A new object file for the UNIX operating system was created with the make(1) system command. This command can be used to recompile the entire system from scratch or to recompile individual source modules and install them in the correct libraries (Ref 4:7, and 5:11). The latter method was used for installing the VG device driver software.

The form of the command used was "make unix60". The input parameter unix60 indicates that only certain source modules were to be recompiled and that the CPU type was 60 (for the PDP11/60).

The make(1) command looks within the current directory for a file named "makefile". This file is used as input to the make(1) command. The file /sys/conf/makefile is the input file needed for regenerating the system (Ref 5:11). A complete listing of this file is given in Appendix E.

The following command stream was used to execute "make unix60" and copy the resulting UNIX object file to /unix.vg.

```
              1.  # cd /sys/conf
              2.  # make unix60
              3.  convert l.s l_i.s
              4.  cp l.s l_i.s
              5.  done converting l_i.s
              6.  as -o l_i.o l_i.s
              7.  as -o mch_i.o mch0.s mch_i.s
              8.  cp ../h/param_i.h ../h/param.h
              9.  cc -c -o c.c
             10.  mv c.o c_i.o
             11.
             12.  The output file will be named unix_i !!!!!
             13.
             14.  ld -o unix_i -x l_i.o mch_i.o c_i.o ../sys/LIB1_i .
/dev/LIB2_i
             15.
             16.  if size of unix_i > 49152 bytes, UNIX IS TOO
BIG !!!!!
             17.
             18.  Size of unix_i is tEXT+DATA+BSS = TOTAL
             19.
             20.  Size unix_i
             21.  33638+1918+13312 = 48868b = 0137344b
             22.  rm *.o
             23.  # cp unix_i /unix.vg
             24.  #
```

First, the current directory was changed to /sys/conf so that the appropriate "makefile" would be used. Next, "make unix60" was invoked on line two. Lines 3-22 are messages printed during successful execution of the command.

The messages indicate what the command was doing. Basically, the files /sys/conf/l.s, /sys/conf/mch0.s, and /sys/conf/mch_i.s were assembled; the file /sys/conf/c.c was compiled; then all the resulting object files were loaded (along with /sys/sys/LIB1_i and /sys/dev/LIB2_i) into an output object file name /sys/conf/unix_i. Next, the size of the object file /sys/conf/unix_i was computed to see if it exceeded 49,152 bytes (the maximum size allowed for a UNIX object file). Finally, on line 22, all the files in the directory /sys/conf that ended in '.o' were removed. This was a cleanup step which removed all intermediate object files created during execution of "make unix60".

On line 23 the UNIX object file /sys/conf/unix_i was copied to the root directory and given the name unix.vg.

Restoring the Changed UNIX Files

The original contents of UNIX files /sys/conf/l.s, /sys/conf/c.c, and /sys/dev/LIB2_i were changed in order to create the new UNIX object file, /unix.vg. A shell procedure, or script file, named /sys/conf/vg_conf.unload was created to restore the original contents of these files after creating the new UNIX object file, /unix.vg.

The script file was created by first using the editor to build a file of system commands, then flagging the file as an executable shell program with the chmod(1) system command (Ref 2:5).

A complete listing of /sys/conf/vg_conf.unload is given here.

```
1.  # cat /sys/conf/vg_conf.unload
2.  echo cp /sys/conf/c.c.save /sys/conf/c.c
3.  cp /sys/conf/c.c.save /sys/conf/c.c
4.  echo
5.  echo cp /sys/conf/l.s.good /sys/conf/l.s
```

```
cp /sys/conf/l.s.good /sys/conf/l.s
echo
echo cp /sys/conf/unix_i.save /sys/conf/unix_i
cp /sys/conf/unix_i.save /sys/conf/unix_i
echo
echo cp /sys/dev/LIB2_i.save /sys/dev/LIB2_i
cp /sys/dev/LIB2_i.save /sys/dev/LIB2_i
echo
echo Finished unloading the configuration for the VG3404 !!!
#
```

This script is executed by typing its file name, /sys/conf/vg_conf.unload, at the system prompt. It restores the original contents of the UNIX files by copying from the appropriate save files.

Rebooting the System from /unix.vg

To use the VG Graphics Display System, the PDP11/60 must be rebooted using the /unix.vg object file. A complete list of commands needed to reboot the system is given in Appendix H. These commands must be executed from the system console. When using this command stream it is assumed that the system is in multi-user mode and that the system console is logged in as the "root" executing a function that monitors system usage. First, the system is taken down from multi-user time sharing mode, then it is rebooted from /unix.vg.

Summary

This chapter presented a complete description of how to install the VG device driver software on the PDP11/60 under UNIX version seven. The device driver software testing methodology is described in the next chapter. All of the test programs and results are included.

IX Software Testing

A few short C-language programs were written to test some of the features of the system. This testing was by no means comprehensive.

The testing methodology used was a combination of program path analysis and "black box" testing. The test programs were written to exercise most of the major program paths of the device driver software. These paths were taken directly from the structure chart in Figure 21 (see page 79 Chapter VI). When the test programs were executed, data was input to the system for which a known output was expected. The actual output was checked against the desired output to verify that the driver software worked properly. With this testing approach, the driver software was treated as a "black box". In other words, the inner workings of the driver software were not observed directly.

The major program paths were tested by writing test programs for each of the VG minor devices. The remainder of this chapter is devoted to a description of the tests performed and their results.

GPU Tests

The open(2), close(2), stty, and gtty system calls were tested on the gpu minor device. The objective was to verify that VG registers could be read and written by a user program and that the GPU could fetch and execute a user display list from the host computer.

The first test performed was to open the GPU minor device, write a value to a VG register, read the same register, print the value read, then close the GPU minor device. The code for the test routine and the execution of the test are listed below.

```
 1.  # cat gputestl.c
 2.  main( )
 3.  { int fdgpu, buf[3];
 4.    fdgpu = open("/dev/gpu",2);
 5.    buf[0] = 012;
 6.    buf[1] = 045;
 7.    stty(fdgpu,buf);
 8.    gtty(fdgpu,buf);
 9.    printf("%o\n", buf[0]);
10.    close(fdgpu);
11.  }
12.  #
13.  #cc gputestl.c
14.  #a.out
15.  45
16.  #
```

Lines 2 through 11 are a listing of the test routine. The routine was compiled on line 13 and executed on line 14. The output was printed on line 15.

In this test the value 45 was written into the VG's picture base object (PBO) register (see lines 5-7), then the PBO register was read to verify that it contained the value 45. The output on line 15 verified that the test was successful. When the PBO register was written (line 7), buf[0] contained the PBO register's address. When the PBO register was read (line 8), buf[0] got changed to the value read.

The same test was performed on the VG's directory (DIR) register. The value 45 was first written to the DIR register, then the DIR register was read. The result was 44 instead of the expected 45. Later, it was discovered that this was not a device driver software error. The VG's DIR register contains logic that converts all odd values to even values. This is done because the directory address stored in the DIR register must begin on a word boundary instead of a byte boundary in computer memory. When even values are written to the DIR register the same values are returned when the register is read.

The next test performed was to verify that the GPU could fetch and execute a user display list stored in the host computer. The display list used was taken from the VG System Reference Manual (Ref 18:4-3). This particular display list contains the instructions needed to draw an equilateral triangle (Ref 18:4-2). The code for the test routine and execution of the test are listed below.

```
# cat tri.c
main( )
{int fdgpu, directry[10], object[50];
 int stack[200], buf[3];
 directry[0] = 01;
 directry[1] = object;
 object[0]   = 01;
 object[1]   = 0140150;
 object[2]   = 0140000;
 object[3]   = 0140000;
 object[4]   = 0040000;
 object[5]   = 0140000;
 object[6]   = 0;
 object[7]   = 0040000;
 object[8]   = 0140000;
 object[9]   = 0140001;
 object[10]  = 0010000;
 fdgpu = open ("/dev/gpu", 2);
 buf[0] = 01;
 buf[1] = stack;
 stty(fdgpu,buf);
 buf[0] = 02;
23.   buf[1] = stack+63;
24.   stty(fdgpu,buf);
25.   buf[0] = 0;
26.   buf[1] = directry;
27.   stty(fdgpu,buf);
28.   buf[0] = 012;
29.   buf[1] = 01;
30.   stty(fdgpu,buf);
31.   buf[0] = 010;
32.   buf[1] = 0401;
33.   stty(fdgpu,buf);
34.   buf[0] = 07;
35.   buf[1] = 0160134;
36.   stty(fdgpu,buf);
37.   close(fdgpu);
38.  }
39.  # cc tri.c
40.  # a.out
41.  GPU interrupt [12] - 130022 73257
42.  #
```

Lines 2 through 38 are the code for the test routine. Lines 5 and 6 set up the directory required for the display list, while lines 7-17 set up the display list itself. Lines 19-21 store the beginning stack address in the VG's stack base address (STB) register. Lines 22-24 store the ending stack address in the VG's stack limit address (SLM) register. Lines 25-27 store the directory address in the VG's directory address (DIR) register. Lines 28-30 store the object number of the base picture in the VG's picture base object (PBO) register. Lines 31-33 load the VG's control (CTL) register, while lines 34-36 load the VG's command (CMD) register. Once the CMD register is loaded, the GPU is directed to fetch and execute the display list stored in the array named "object".

The test routine "tri.c" was compiled and executed on lines 39-41. The result was an interrupt generated by the GPU with state code 12 (see line 41). State code 12 means that an invalid picture base object or directory structure caused the interrupt (Ref 18: Appendix B2). One probable cause of this error is an invalid base address stored in the Hardware Interface's Base Address Register (BAR). This would cause all virtual addresses to be mapped to incorrect physical addresses. If this was the problem, then the GPU used an erroneous physical address to fetch the user's directory information and found an invalid directory structure stored there.

The way to discover if the Interface's BAR is being loaded with the correct address is to write a user program that performs the same address mapping that the Interface performs (Ref 19:13). When performing the address mapping, use the same base address that the driver software loads into the Interface's BAR. The program should first store some predetermined value in a known location. Next the program maps the known location's virtual address to a physical address using the same base address and address mapping algorithm used by the Hardware Interface. Finally, the program fetches the contents of the calculated physical address to see if it is the predetermined value that was stored there in the beginning. If it is, then the error occuring in the driver software was probably not caused by the Hardware Interface's address mapping. On the other hand, if the value fetched is not the same as the value stored then the base address used during the address mapping was erroneous. If this is the case, then the User Instruction Space Address (UISA) Registers have probably been changed between UNIX versions six and seven. In that case, the driver software would have to be changed to load the Interface's BAR from the correct UISA register.

Data Tablet Tests

The open(2), read(2), and close(2) system calls were tested on the data tablet minor device. The objective was to verify that a user program could read the VG's data tablet registers and that a user program could select which of the four types of data tablet interrupts it would recognize.

The first test performed was to open the data tablet minor device, mask out all data tablet interrupts except those generated by the pressure switch on the data tablet stylus, read the data tablet minor device, then close it. The code for the test routine and the execution of the test are listed below.

```
# cat dtb.c
main( )
{
int fdgpu, fddtb, n, buf[50];
fdgpu = open("/dev/gpu",2);
fddtb = open("/dev/dtb",2);
buf[0] = -5;
buf[1] = 01;
stty(fdgpu,buf);
n = 0;
while (n<1) n=read(fddtb, &buf, 1);
printf("Flag = %o, X = %d, Y = %d\n", buf[1], buf[2]);
close(fddtb);
close(fdgpu);
}
#
#
# cc dtb.c
# a.out
Flag = 1, X = 3, Y = 41
# a.out
```

```
22.  Flag = 1, X = -377, Y = 441
23.  # a.out
24.  Flag = 1, X = -408, Y = -319
25.  # a.out
26.  Flag = 1, X = 369, Y = -318
27.  #
```

Lines 5 and 6 open the GPU and data tablet minor devices. Lines 7 through 9 select the pressure switch interrupt (PRS) only. Lines 11 and 12 read and print one X-Y coordinate pair from the data tablet and the type of interrupt that generated the pair. Lines 13 and 14 close the data tablet and gpu minor devices.

Lines 19 through 26 contain the results of four different executions of the test routine. For each test, the author used the data tablet stylus to generate all the different types of interrupts available on the data tablet, i.e., XOS, YOS, PNN, and PRS (Ref 17:2-83). The phrase "Flag = 1" on each line of the output (lines 20, 22, 24, and 26) verifies that only the X-Y coordinate pairs generated by the PRS interrupt were passed to the test program.

For the next test, lines 7-9 of the data tablet test program were omitted. This meant that X-Y coordinate pairs generated by any of the four data tablet interrupts could be read by the test program. Four different executions of this test program and the resulting output are listed below.

```
1.  # cc dtb.c
2.  # a.out
3.  Flag = 2, X=11, Y=75
4.  # a.out
5.  Flag = 1, X=413, Y=474
6.  # a.out
```

```
 7.  Flag = 4, X=125, Y=512
 8.  # a.out
 9.  Flag = 8, X=512, Y=-305
10.  #
```

In the first execution (lines 2-3), "Flag = 2" indicates that the X-Y coordinate pair was generated by a PNN interrupt. For the second, third, and fourth executions, the X-Y coordinate pairs were generated by the PRS, YOS, and XOS interrupts respectively.

For all of the above data tablet tests the device driver software performed correctly. Therefore, the objective of the data tablet tests was met.

Keyboard Tests

The open(2), read(2), and close(2) system calls were tested on the VG's alphanumeric keyboard minor device. The objective was to verify that a user program could read data from the VG's alphanumeric keyboard input device.

A short test program was written to read and print out fourteen characters from the VG's alphanumeric keyboard. The test program and three different executions of the test are listed below.

```
# cat kbd.c
main( )
{ int i, fdgpu, fdkbd, n, buf[50];
  fdgpu = open("/dev/gpu",2);
  fdkbd = open("/dev/kbd",2);
  for (i=1; i<=14; i++) {
     n=o;
     while (n<1) n=read(fdkbd,&buf,1);
     printf("%c",buf[0]);
  }
  printf("\n");
```

```
12.        close(fdkbd);
13.        close(fdgpu);
14.     }
15.     # cc kbd.c
16.     # a.out
17.     this is a test
18.     # a.out
19.     This is a TEST
20.     # a.out
21.     123456789{! ? < >
22.     #
```

Lines 2-14 are the test program while lines 16-21 are three different executions of the test program. The first and second executions (lines 16-19) verified that both upper and lower case letters were read successfully. The third execution (lines 20-21) verified that numeric and other special characters were read successfully. Therefore, the objective of this test was met.

<u>Function Switch Box Tests</u>

The open(2), read(2), and close(2) system calls were tested on the VG's function switch box minor device. The objective was to verify that a user program could read values from the VG's function switch box input device.

A test program was written to read one value from the function switch box. The test program and four executions of the test are listed below.

```
# cat fss.c
main( )
{
int fdgpu, fdfss, n, buf;
fdgpu = open("/dev/gpu",2);
fdfss = open("/dev/fss",2);
n = 0;
while (n<1) n=read(fdfss, &buf, 1);
```

```
 9.  printf("function switch = %d\n", buf);
10.  close(fdfss);
11.  close(fdgpu);
12.  }
13.  # cc fss.c
14.  # a.out
15.  function switch = 1
16.  # a.out
17.  function switch = 15
18.  # a.out
19.  function switch = 25
20.  # a.out
21.  function switch = 31
22.  #
```

Line 2-12 are a listing of the test program. Lines 14-21 contain four different executions of the test program. For each execution of the test program the author verified that the value printed was the number of the function switch that was depressed. Therefore the objective of the function switch box tests was met.

Summary

Most of the major features of the system were tested. Except for the direct memory access test performed on the GPU minor device, all tests performed on the device driver software were successful. This testing concluded the author's research. Conclusions and recommendations are presented in the next chapter.

X Conclusions and Recommendations

The UNIX operating system provides a straight-forward interface to peripheral device driver software. This interface allows for the addition of any number of peripheral devices to the system. The limiting factor is the amount of memory available for the operating system. This was a major problem with AFIT's PDP 11/60. Space was so limited that the VG graphics display system could only be used while the PDP 11/60 was in single user mode. This unacceptable situation can be remedied with the newer version of UNIX which has a memory overlay capability. This capability will allow the operating system to support more device drivers.

The differences between UNIX versions six and seven were transparent to the common user but not to the systems programmer. Therefore, a computer installation that upgrades to a later version of UNIX may have to convert some of their device driver software. Many changes had to be made to McCallum's original driver before it would run under UNIX version seven.

The fact that UNIX is written in a High Order Language (HOL) such as "C" is a real asset. This aids the systems programmer immensely in understanding and maintaining the system. It is also very convenient to be able to write the device driver software in the same HOL. The C programming language has many features which lend to systems programming, e.g., pointers and structures.

McCallum's design for the VG device driver was straightforward and easy to understand. He used a top down modular approach which allows for easy expansion of the driver software. This was shown by the easy addition of the data tablet minor device to the driver software.

The apparent problem with direct memory access must be solved before the system is useful. First, the cause of the problem must be identified, then corrected. A probable cause of the problem and a possible solution were identified in Chapter IX.

Many worthwhile projects could stem from the research in this thesis. One project would be to implement a time-out capability when reading from the VG input devices. In other words, if no input data is available when a user program reads a VG input device then the user program should be put to "sleep" for a short time to wait for data to be input from the device.

The author attempted to implement the time-out feature using the alarm(2), pause(2), and signal(2) system calls. The attempt was aborted when it was discovered that the header file required by the signal(2) system call (signal.h) was not available on AFIT's system. The following three lines of code show how the time-out would have worked using the three system calls.

```
1.  alarm(n);
2.  pause( );
3.  (*signal(SIGALRM,SIG_IGN))( );
```

The author intended to have the device driver execute this code if no data was available when a user program attempted to read a VG input device. Line 1 tells UNIX to send an alarm signal to this process after n seconds have elapsed. Line 2 causes the driver to stop execution to wait for a signal. Line 3 catches the alarm signal sent by UNIX after the n seconds have elapsed. After the alarm signal is caught the device driver resumes execution. This would have been an easy way to implement the time-out feature. Since the file /sys/h/signal.h was not present on the system, the time-out feature could not be done with the alarm(2), pause(2), and signal(2) system calls. Nevertheless, a time-out feature could be programmed in other ways.

Another possible project would be to enhance the input capabilities of the VG's alphanumeric keyboard input device. Currently the device driver only supports the "raw" mode of input from the keyboard. That is, no special meaning is assigned to any input character received from the VG's keyboard. The device driver could be changed to support "cooked" input from the VG keyboard. That is, control characters input from the VG's keyboard could be detected by the device driver and handled in a special way. Another project in this area would be to echo the keyboard characters to the VG's display.

Perhaps the most worthwhile project is the implementation of a high level device independent graphics software package on the system. McCallum's level two graphics software (Ref 12) is readily available. It may have to be modified a little to be compatible with UNIX version seven's version of device driver software.

Another graphics software system such as Lawrence Livermore's Grafcore/Graflib (Ref 6) could also be implemented. In this case a BASELIB would have to be generated to define the UNIX/Graflib interface. Next a filter would have to be written to convert the device independent display list that Graflib produces into a display list that can be processed by the VG display system.

Many more worthwhile thesis projects could be undertaken to develop AFIT's computer graphics capabilities. The field is wide open and the options are virtually limitless.

Bibliography

1. Bell Telephone Laboratories. UNIX Time-Sharing System: UNIX Programmer's Manual, 1. Murray Hill: Bell Telephone Laboratories, Inc., January 1979.

2. Bourne, S. R. "An Introduction to the UNIX Shell," UNIX Programmer's Manual, 2A. Murray Hill: Bell Telephone Laboratories, Inc., January 1979.

3. Bourne, S. R. "The UNIX Shell," The Bell System Technical Journal, 57 (6): 1971-1990 (July-August 1978).

4. Haley, Charles B. and Dennis M. Ritchie. "Regenerating System Software," UNIX Programmer's Manual, 2A. Murray Hill: Bell Telephone Laboratories, Inc., January 1979.

5. Haley, Charles B. and Dennis M. Ritchie. "Setting Up UNIX - Seventh Edition," UNIX Programmer's Manual, 2A. Murray Hill: Bell Telephone Laboratories, Inc., January 1979.

6. Keller, Pete, et al. GRAFLIB Reference Manual. Livermore: Lawrence Livermore Laboratory, October 1980.

7. Kernighan, Brian W. and Dennis M. Ritchie. The C Programming Language. Englewood Cliffs: Prentice-Hall, Inc., 1978.

8. Kernighan, Brian W. "A Tutorial Introduction to the UNIX Text Editor," UNIX Programmer's Manual, 2A: 54-64. Murray Hill: Bell Telephone Laboratories, Inc., January 1979.

9. Kernighan, Brian W. "UNIX for Beginners - Seventh Edition," UNIX Programmer's Manual, 2A: Murray Hill: Bell Telephone Laboratories, Inc., January 1979.

10. Lions, J. A Commentary on the UNIX Operating System. Kensington: Department of Computer Science, The University of New South Wales, June 1977.

11. Lions, J. UNIX Operating System Source Code, Level Six. Kensington: Department of Computer Science, The University of New South Wales, June 1977.

12. McCallum, Douglas Roland. A Machine-Independent Interactive Computer Graphics System. MA thesis. Austin, Texas: The University of Texas at Austin, May 1980.

13. Ritchie, D. M. and K. Thompson. "The UNIX Time-Sharing System," UNIX Programmer's Manual, 2A: 23-38. Murray Hill: Bell Telephone Laboratories, Inc., January 1979.

14. Ritchie, Dennis M. "The UNIX I/O System," UNIX Programmer's Manual, 2A. Murry Hill: Bell Telephone Laboratories, Inc., January 1979.

15. SIGGRAPH. "Status Report of the Graphic Standards Planning Committee," Computer Graphics, A Quarterly Report of SIGGRAPH-ACM, 13 (3). (August 1979).

16. Thompson, K. "UNIX Implementation," UNIX Programmer's Manual, 2A. Murray Hill: Bell Telephone Laboratories, Inc., January 1979.

17. Vector General. Graphics Display System Model 3404 Programming Concepts Manual. Publication number 113489. Woodland Hills: Vector General Inc., July 1978.

18. Vector General. Graphics Display System Model 3404 System Reference Manual. Publication number M110700REF. Woodland Hills: Vector General Inc., August 1978.

19. Vector General. PDP11 Interface Specifications. DE41 reference manual. Woodland Hills: Vector General Inc.

20. Vector General. Series 3400 Technical Manual, Volume 1. Publication number M110700. Woodland Hills: Vector General Inc., March 1978.

Appendix A: Listings of UNIX Source Files /sys/h/proc.h and /sys/h/user.h

The two UNIX Source Files contained in this appendix were printed on the PDP11/60 system. The following two command lines were invoked to print the files on a teletype terminal.

```
1.  #  printit </sys/h/proc.h
2.  #  printit </sys/h/user.h
```

The program "printit" was written to print the input file with line numbers added. The source code for this program is listed below.

```
#  printit <printit.c
    01   #include "/sys/h/stdio.h"
    02   #define MAXLINE 133
    03   main()
    04   {       register int i;
    05           char *temp[133];
    06
    07           for (i=1;fgets(temp,MAXLINE,stdin);
                 i++) {
    08                   fprintf(stdout,"%5.5d   ",i);
    09                   fputs(temp,stdout);
   010           }
   011   }
#
```

The prinit program was used to print all of the source file listings in Appendices A-E.

```
/*
 * One structure allocated per active
 * process. It contains all data needed
 * about the process while the
 * process may be swapped out.
 * Other per process data (user.h)
 * is swapped with the process.
 */
struct  proc {
        char    p_stat:
        char    p_flag:
        char    p_pri;          /* priority, negative is high */
        char    p_time;         /* resident time for scheduling */
        char    p_cpu;          /* cpu usage for scheduling */
        char    p_nice;         /* nice for cpu usage */
        short   p_sig;          /* signals pending to this process */
        short   p_uid;          /* user id, used to direct tty signals */
        short   p_pgrp:         /* name of process group leader */
        short   p_pid:          /* unique process id */
        short   p_ppid;         /* process id of parent */
        short   p_addr;         /* address of swappable image */
        short   p_size;         /* size of swappable image (clicks) */
        caddr_t p_wchan;        /* event process is awaiting */
        struct text *p_textp;   /* pointer to text structure */
        struct proc *p_link:    /* linked list of running processes */
        int     p_clktim:       /* time to alarm clock signal */
};

extern struct proc proc[];      /* the proc table itself */

/* stat codes */
#define SSLEEP  1               /* awaiting an event */
#define SWAIT   2               /* (abandoned state) */
```

```
#define SRUN    3               /* running */
#define SIDL    4               /* intermediate state in process creation */
#define SZOMB   5               /* intermediate state in process termination */
#define SSTOP   6               /* process being traced */

/* flag codes */
#define SLOAD   01              /* in core */
#define SSYS    02              /* scheduling process */
#define SLOCK   04              /* process cannot be swapped */
#define SSWAP   010             /* process is being swapped out */
#define STRC    020             /* process is being traced */
#define SWTED   040             /* another tracing flag */
#define SULOCK  0100            /* user settable lock in core */

/*
 * parallel proc structure
 * to replace part with times
 * to be passed to parent process
 * in ZOMBIE state.
 */
struct  xproc {
        char    xp_stat;
        char    xp_flag;
        char    xp_pri:         /* priority, negative is high */
        char    xp_time:        /* resident time for scheduling */
        char    xp_cpu;         /* cpu usage for scheduling */
        char    xp_nice;        /* nice for cpu usage */
        short   xp_sig:         /* signals pending to this process */
        short   xp_uid·         /* user id, used to direct tty signals */
        short   xp_pgrp:        /* name of process group leader */
        short   xp_pid;         /* unique process id */
        short   xp_ppid;        /* process id of parent */
        short   xp_xstat;       /* Exit status for wait */
```

```
            time_t  xp_utime;       /* user time, this proc */
            time_t  xp_stime:       /* system time, this proc */
    };
```

```
Dec  8 14:46 1981       Listing of Source File /sys/h/user.h          Page 1

/*
 * The user structure.
 * One allocated per process.
 * Contains all per process data
 * that doesn't need to be referenced
 * while the process is swapped.
 * The user block is USIZE*64 bytes
 * long; resides at virtual kernel
 * loc 140000; contains the system
 * stack per user; is cross referenced
 * with the proc structure for the
 * same process.
 */

#define EXCLOSE 01

struct  user
{
        label_t u_rsav;                 /* save info when exchanging stacks */
        int     u_fper;                 /* FP error register */
        int     u_fpsaved;              /* FP regs saved for this proc */
        struct {
                int     u_fpsr;         /* FP status register */
                double  u_fpregs[6];    /* FP registers */
        } u_fps;
        char    u_segflg;               /* IO flag: 0:user D; 1:system; 2:user I */
        char    u_error;                /* return error code */
        short   u_uid;                  /* effective user id */
        short   u_gid;                  /* effective group id */
        short   u_ruid;                 /* real user id */
        short   u_rgid;                 /* real group id */
        struct proc *u_procp;           /* pointer to proc structure */
        int     *u_ap;                  /* pointer to arglist */
        union {                                 /* syscall return values */
                struct  {
                        int     r_val1;
                        int     r_val2;
                };
                off_t   r_off;
                time_t  r_time;
        } u_r;
        caddr_t u_base;                         /* base address for IO */
        unsigned int u_count;                   /* bytes remaining for IO */
        off_t   u_offset;                       /* offset in file for IO */
        struct inode *u_cdir;                   /* pointer to inode of current directory */
        struct inode *u_rdir;                   /* root directory of current process */
        char    u_dbuf[DIRSIZ];                 /* current pathname component */
        caddr_t u_dirp;                         /* pathname pointer */
        struct direct u_dent;                   /* current directory entry */
        struct inode *u_pdir;                   /* inode of parent directory of dirp */
        int     u_uisa[16];                     /* prototype of segmentation addresses */
        int     u_uisd[16];                     /* prototype of segmentation descriptors */
        struct file *u_ofile[NOFILE];           /* pointers to file structures of open files */
        char    u_pofile[NOFILE];               /* per-process flags of open files */
        int     u_arg[5];                       /* arguments to current system call */
        unsigned u_tsize;                       /* text size (clicks) */
        unsigned u_dsize;                       /* data size (clicks) */
        unsigned u_ssize;                       /* stack size (clicks) */
        label_t u_qsav;                         /* label variable for quits and interrupts */
        label_t u_ssav;                         /* label variable for swapping */
        int     u_signal[NSIG];                 /* disposition of signals */
        time_t  u_utime;                        /* this process user time */
        time_t  u_stime;                        /* this process system time */
        time_t  u_cutime;                       /* sum of childs' utimes */
        time_t  u_cstime;                       /* sum of childs' stimes */
        int     *u_ar0;                         /* address of users saved R0 */
        struct {                                /* profile arguments */
                short   *pr_base;               /* buffer base */
                unsigned pr_size;               /* buffer size */
                unsigned pr_off;                /* pc offset */
                unsigned pr_scale;              /* pc scaling */
        } u_prof;
        char    u_intflg;                       /* catch intr from sys */
        char    u_sep;                          /* flag for I and D separation */
        struct tty *u_ttyp;                     /* controlling tty pointer */
        dev_t   u_ttyd;                         /* controlling tty dev */
        struct {                                /* header of executable file */
                int     ux_mag;                 /* magic number */
                unsigned ux_tsize;              /* text size */
                unsigned ux_dsize;              /* data size */
                unsigned ux_bsize;              /* bss size */
                unsigned ux_ssize;              /* symbol table size */
                unsigned ux_entloc;             /* entry location */
                unsigned ux_unused;
                unsigned ux_relflg;
        } u_exdata;
        char    u_comm[DIRSIZ];
        time_t  u_start;
        char    u_acflag;
        short   u_fpflag;                       /* unused now, will be later */
        short   u_cmask;                        /* mask for file creation */
        int     u_stack[1];
                                                /* kernel stack per user
                                                 * extends from u + USIZE*64
                                                 * backward not to reach here
                                                 */
};

extern struct user u;
```

```

/* u_error codes */
#define EPERM   1
#define ENOENT  2
#define ESRCH   3
#define EINTR   4
#define EIO     5
#define ENXIO   6
#define E2BIG   7
#define ENOEXEC 8
#define EBADF   9
#define ECHILD  10
#define EAGAIN  11
#define ENOMEM  12
#define EACCES  13
#define EFAULT  14
#define ENOTBLK 15
#define EBUSY   16
#define EEXIST  17
#define EXDEV   18
#define ENODEV  19
#define ENOTDIR 20
#define EISDIR  21
#define EINVAL  22
#define ENFILE  23
#define EMFILE  24
#define ENOTTY  25
#define ETXTBSY 26
#define EFBIG   27
#define ENOSPC  28
#define ESPIPE  29
#define EROFS   30
#define EMLINK  31
```

```
#define EPIPE   32
#define EDOM    33
#define ERANGE  34
```

Appendix B: Listing of UNIX Source File /sys/conf/l.s.vg

The UNIX source file /sys/conf/l.s.vg contains the system call trap vector (line 31) and the VG's interrupt vector (lines 60-61 and 83-84).

```
/
/       Edited to include the interrupt vector for the
/       Vector General 3404 Graphics Display System.
/
/       Since the dh11 driver was removed from LIB2_i
/       to make room for the VG driver, the DH11 and DM11
/       interrupt vectors have been removed from this file.
/
/
/ low core

.data
ZERO:

br4 = 200
br5 = 240
br6 = 300
br7 = 340

. = ZERO+0
        br      1f
        4

/ trap vectors
        trap; br7+0.            / bus error
        trap: br7+1.            / illegal instruction
        trap; br7+2.            / bpt-trace trap
        trap; br7+3.            / iot trap
        trap; br7+4.            / power fail
        trap: br7+5.            / emulator trap
        start;br7+6.            / system  (overlaid by 'trap')

. = ZERO+40
```

```
.globl  start, dump
1:      jmp     start
        jmp     dump


. = ZERO+60
        klin; br4
        klou; br4

. = ZERO+100
        kwlp: br6
        kwlp: br6

. = ZERO+114
        trap: br7+10.           / 11/70 parity

. = ZERO+210
        hkio: br5

. = ZERO+240
        trap; br7+7.            / programmed interrupt
        trap; br7+8.            / floating point
        trap; br7+9.            / segmentation violation

/ floating vectors

. = ZERO+374
        vgint; br7              / Vector General interrupt vector

/////////////////////////////////////////////////////
/                  interface code to C
/////////////////////////////////////////////////////

```

```
.text
.globl  call, trap

.globl  _klrint
klin:   jsr     r0,call; jmp  klrint
.globl  _klxint
klou:   jsr     r0,call; jmp _klxint

.globl  _clock
kwlp:   jsr     r0,call; jmp _clock


.globl  _hkintr
hkio:   jsr     r0,call; jmp _hkintr


.globl  _vgint
vgint:  jsr     r0,call; jmp _vgint
```

Appendix C: Listing of UNIX Source File /sys/conf/c.c.vg

The UNIX source file /sys/conf/c.c.vg contains the system character device switch table (cdevsw). The cdevsw table contains the addresses of the VG major device routines (line 77).

```
/**************************************************************/
/* Edited to include the Vector General 3404 Graphics Device. */
/* It has been added to the cdevsw table as major device 22   */
/*                                                            */
/*                                                            */
/* The Vector General 3404 device driver would not fit in     */
/* the system, so the device driver dh = 4 was removed to     */
/* make room.  This means that you can only use the           */
/* Vector General Graphics Device while in single user mode.  */
/* Multi-user mode is not supported.                          */
/**************************************************************/


#include "../h/param.h"
#include "../h/systm.h"
#include "../h/buf.h"
#include "../h/tty.h"
#include "../h/conf.h"
#include "../h/proc.h"
#include "../h/text.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../h/file.h"
#include "../h/inode.h"
#include "../h/acct.h"

int     nulldev()·
int     nodev();
int     hkstrategy():
struct  buf     hktab:
struct  bdevsw  bdevsw[] =
{
        nodev, nodev, nodev, 0, /* rk = 0 */
```

```
		nodev, nodev, nodev, 0, /* rp = 1 */
		nodev, nodev, nodev, 0, /* rf = 2 */
		nodev, nodev, nodev, 0, /* tm = 3 */
		nodev, nodev, nodev, 0, /* tc = 4 */
		nodev, nodev, nodev, 0, /* hs = 5 */
		nodev, nodev, nodev, 0, /* hp = 6 */
		nodev, nodev, nodev, 0, /* ht = 7 */
		nodev, nodev, nodev, 0, /* rl = 8 */
		nulldev, nulldev, hkstrategy, &hktab,   /* hk = 9 */
		nodev, nodev, nodev, 0, /* ts = 10 */
		0
	};

	int	klopen(), klclose(), klread(), klwrite(), klioctl();
	int	mmread(), mmwrite():
	int	syopen(), syread(), sywrite(), sysioctl();
	int	hkread(), hkwrite():
	int	vgopen(), vgclose(), vgread(), vgwrite(), vgioctl();

	struct	cdevsw  cdevsw[] =
	{
		klopen, klclose, klread, klwrite, klioctl, nulldev, 0,  /* console = 0 */
		nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* pc = 1 */
		nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* lp = 2 */
		nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* dc = 3 */
		nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* dh = 4 */
		nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* dp = 5 */
		nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* dj = 6 */
		nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* dn = 7 */
		nulldev, nulldev, mmread, mmwrite, nodev, nulldev, 0,  /* mem = 8 */
		nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* rk = 9 */
		nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* rf = 10 */
		nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* rp = 11 */
```

```
                nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* tm = 12 */
                nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* hs = 13 */
                nodev  nodev, nodev, nodev, nodev, nulldev, 0, /* hp = 14 */
                nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* ht = 15 */
                nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* du = 16 */
                syopen, nulldev, syread, sywrite, sysioctl, nulldev, 0, /* tty = 17 */
                nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* rl = 18 */
                nulldev, nulldev, hkread, hkwrite, nodev, nulldev, 0,   /* hk = 19 */
                nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* ts = 20 */
                nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* dz = 21 */
                vgopen, vgclose, vgread, vgwrite, vgioctl, nulldev, 0, /* vg = 22 */
                0
        };

        int     ttyopen(), ttyclose(), ttread(), ttwrite(), ttyinput(), ttstart();
        struct  linesw  linesw[] =
        {
                ttyopen, nulldev, ttread, ttwrite, nodev, ttyinput, ttstart, /* 0 */
                0
        };
        int     rootdev = makedev(9, 0);
        int     swapdev = makedev(9, 1):
        int     pipedev = makedev(9, 0):
        int     nldisp = 1;
        daddr_t swplo   = 0;
        int     nswap   = 8778;

        struct  buf     buf[NBUF];
        struct  file    file[NFILE];
        struct  inode   inode[NINODE]:
        int     mpxchan():
        int     (*ldmpx)() = mpxchan;
        struct  proc    proc[NPROC];
```

```
struct  text    text[NTEXT];
struct  buf     bfreelist;
#if SID
struct  acct    acctbuf;
struct  inode   *acctp;
#endif
```

Appendix D: Listings of Driver Source Files /sys/h.vg.h and /sys/dev/vg.c

The VG device driver source code is located in two files; /sys/h/vg.h and /sys/dev/vg.c. These files contain the final version of the device driver software.

```
Dec 20 02:23 1981        Driver Source File /sys/h/vg.h           Page 1

#define dctl    00400
#define dcust   00402
#define ctl     00010
#define rbustc  01400
#define reset   00010
#define rb_mask 060000


#define vg_oip  040000
#define vg_iip  0100000
#define r_change        02000
#define vg_rqi  vg_iip
#define rbu_b   01401
#define dtx     01600
#define dty     01601
#define dtb     01602
#define kbd     01607
#define fss     01604
#define vg_init 020000

#define OPEN    1
#define SLEEP   2
#define RUNNING 040
#define WAITING 0100
#define GO      040000
#define PICHE   020000
#define NPIC    0100000
#define VG_CONT 0163402
#define VG_DATA 0163404
#define VG_BAR  0163406
#define VG_STAT 0163400
#define APR     0177640
```

```


/*      interrupt vectors       */

#define DATABLET        040
#define KEYBOARD        047
#define FUNCTION        044
#define RBU_UNIT        004
#define GPU_UNIT        002
#define GP_BUS          000

#define intenable       010000
#define intack          004000
#define rbumar          01406
#define rbudat          01407
#define cmd     0007
#define stat    0011
















```

```
/*      Handler for Vector General Processor
        This handler must take several minor devices into consideration/
        gp, dt, kb, fs                                                   */


#include "../h/param.h"
#include "../h/buf.h"
#include "../h/conf.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../h/tty.h"
#include "../h/proc.h"
#include "../h/vg.h"

int dtintmask;


/* An array handling all minor device information */

struct vgstruc {
        struct clist io;
        int status;
        int *vg_procp;
        } vgunit[4];


struct {   char lobyte;    char hibyte;    };


struct {    char d_minor;    char d_major;    };


struct {    int reg;    }
```

```
vgsgtty(dev,command)
int     dev, command;
{
  int v[3];
  register *up, *vp;
  vp = v;
  up = u.u_arg[2];
  switch (command) {
  case TIOCGETP:
        *vp = fuiword(up);
        *vp = PIN(*vp);
        suiword(up, *vp);
        break;
  case TIOCSETP:
        spl7();
        *vp = fuiword(up);
        *(vp+1) = fuiword(++up);
        *(vp+2) = fuiword(++up);
        switch (*vp){
        case -1:
                POUT(rbumar, *++vp);
                POUT(rbudat, *++vp);
                break;
        case -2:
                RBURSET();
                break;
        case -3:
                gpwait();
                break;
        case -4:
                gpurestart();
                break;

        case -5:
                dtintmask = *(vp+1);
                break;
        default:
                POUT(*vp, *(vp+1));
                break;
        }
        spl0();
        break;
  default:
printf("unrecognized cmd\n");
        return;
        break;
  }
}



fskbdtsgtty(dev,command)
int dev, command;
{
  int s;
  register *up;
  up = u.u_arg[2];
  switch(command) {
  case TIOCGETP:
        s = vgunit[dev.d_minor].status;
        suiword(up, s);
        break;
  case TIOCSETP:
        s = fuiword(up);
        vgunit[dev.d_minor].status = OPEN | s;
        break;
```

```
    default:
         return;
         break;
    }
}






/*
 * commonly used routines
 * POUT and PIN perform the function POUT or PIN described in VG manual
 */



POUT(REGISTER, VALUE)
int REGISTER, VALUE;
{ VG_CONT->reg = (REGISTER & 01777) | r_change | intenable;
  VG_DATA->reg = VALUE;
  while (VG_CONT->reg & vg_oip) /* wait until done */;
}       /* POUT */



PIN(REGISTER)
int REGISTER;
{ VG_CONT->reg = (REGISTER & 01777) | r_change | vg_rqi | intenable;
  while (VG_CONT->reg & vg_iip) /* wait for done */;
  return VG_DATA->reg;
}       /* PIN */
```

```
RBURSET()
{
        POUT(rbustc,reset);
        while ((PIN(rbustc) & rb_mask) != rb_mask) /* wait */ ;
}






/*      VG data tablet handler     */


dtopen()
{       dtintmask=017;
        dtstart();

}



dtclose()
{  extern struct cdevsw vgdev[];
        POUT(dtb, 0);                     /* disable interrupts */
        while (getc(&vgunit[1].io) >= 0) ; /* flush buffer */
}


dtstart()
{  extern struct cdevsw vgdev[];
        POUT(dtb, 01);                    /* enable interrupts */
}
```

```
dtwrite()
{       u.u_error = EIO;     }


dtread()
{  register int c;

   spl0();

   u.u_count = u.u_count * 06;

   while (u.u_count && (vgunit[1].io.c_cc > 0)) {
        c = getc(&vgunit[1].io);
        passc(c);
   }
}


dtintr()
{  int dtbx, dtby, status;
   status = PIN(dtb)>>1;
   if (status &= dtintmask) {
        dtbx = PIN(dtx) >> 06;
        dtby = PIN(dty) >> 06;
        putc(status.lobyte, &vgunit[1].io);
        putc(status.hibyte, &vgunit[1].io);
        putc(dtbx.lobyte, &vgunit[1].io);
        putc(dtbx.hibyte, &vgunit[1].io);
        putc(dtby.lobyte, &vgunit[1].io);
        putc(dtby.hibyte, &vgunit[1].io);
   }
   dtstart();
}
```

```
/*      VG keyboard handler     */



kbopen()
{       kbstart();
}




kbclose()
{  extern struct cdevsw vgdev[];
        POUT(kbd,0);
        while(getc(&vgunit[2].io) >= 0) ;/*flush buffer */
}




kbstart()
{
  extern struct cdevsw vgdev[];
        POUT(kbd,040000);       /* enable interrupts */
}




kbwrite()
{
        u.u_error = EIO;
}
```

```
kbread()
{ register int C;
  spl0();

   while (u.u_count){
        while (u.u_count && (C = getc(&vgunit[2].io)) != -1){
                if (C == '\015') C = '\n';
                passc(C);
                switch ( C ){

                case '\n':
                case '\004':
                     return;
                default:        ;
                }
        }
   }
}




kbintr()
{ register int C;
  extern struct cdevsw vgdev[];
  C = PIN(kbd) & 0377;
  putc(C,&vgunit[2].io);
  kbstart();
}
```

```
/*
 * fss device
 * this is the function switch box input device
 */


fsopen()
{
        fsstart();
}



fsclose()
{
        POUT(fss+2,0);
        while(getc(&vgunit[3].io) >= 0); /* flush buffer */
}



fsstart()
{
        POUT(fss+2,05005);
}



fswrite()
{
        u.u_error=EIO;
}

```

```
fsread()
{ unsigned DATA;
  register unsigned C, BIT;
  int           Q, COUNT;
  extern struct cdevsw vgdev[];

  spl0();

  while ( u.u_count && (vgunit[3].io.c_cc != 0)) {
        Q = getc(&vgunit[3].io);
        DATA.lobyte = getc(&vgunit[3].io);
        DATA.hibyte = getc(&vgunit[3].io);

        /* convert to an integer between 0 and 15
         * then put into proper range of values to
         *  get 0 through 31 as values depending on first
         * byte of data.
         */

        C = DATA;
        BIT = 0100000;
        COUNT = 0;
        while (!( C & BIT) && (COUNT < 16)){
                                            BIT >>= 1;
                                            COUNT ++;
                                       }
        passc(Q ? COUNT+16 : COUNT);
  }
}
```

```
fsintr()
{ int C,Q;

  Q = 0;

  if (PIN(fss+2) & 01000)
        C = PIN(fss);
  else{
        C = PIN(fss+1);
        Q = 1;
  }
  putc(Q.lobyte,&vgunit[3].io); /* flag for upper or lower range */
  putc(C.lobyte,&vgunit[3].io);
  putc(C.hibyte,&vgunit[3].io);
  fsstart();
}
```

```
/*
 * GPU device
 * this handles the gpu portion of the VG
 * system dependencies are mainly the way to lock a process into core
 *
 * NOTE: process must be locked in core and be contiguous in memory
 */




gpopen(dev,flag)
{
  int Text, *ap;

  /*    lock the process into core in order to prevent swapping */

  u.u_procp->p_flag |= (SSYS|SLOCK);

  VG_CONT->reg= 020000; /* initialize whenever GPU opened */

  ap = APR;
  if(( Text = u.u_procp->p_textp) != NULL) /* then we don't want the zeroth
                                            * page of memory which may not
                                            * be contiguous in space with
                                            * the rest of process
                                            */
        ap += ((Text+127)>>7)-1;

  VG_BAR->reg = *ap;                       /* set up base address register */
}


```

```
rbread()
{  int DATA;
   if (u.u_count&1) u.u_count--;
   if (u.u_base &1) u.u_base --;

   while (u.u_count){
      POUT(rbumar,u.u_offset);
      DATA = PIN(rbudat);
      passc(DATA.lobyte);
      passc(DATA.hibyte);
      u.u_offset++;
   }
}


gpwrite()
{
      u.u_error=EIO;
}





gpclose()
{
      POUT(cmd,000000);
      POUT(dctl,040000);
      u.u_procp->p_flag &= ~(SSYS|SLOCK);
}
```

```
gpintr()
{
register int state;
extern gpurestart();

  switch (state = PIN(stat)){

  case 001:
  case 017:
        vgunit[0].status &= ~RUNNING;
        if (vgunit[0].status & SLEEP)
           wakeup(&vgunit[0].io);
        else{
           vgunit[0].status |= WAITING;
           gpurestart();
        }
           break;

  case 022:
  case 023:
           /*
            * trace mode -- WARNING: Hardware problems, DO NOT USE
            */
           psignal(vgunit[0].vg_procp,5);
           break;

  default:

  printf("GPU interrupt [%o] - %o %o\n",PIN(stat),PIN(04),PIN(013));
  VG_CONT->reg = 02000;         /* initialize on a panic */
  break;
  }
}
```

```
/*
 * VG the generic device for the Vector General
 *
 * This is to avoid having a lot of major device numbers and allowing
 * all sub devices to be minor devices
 */



struct cdevsw vgdev[] =
        {&gpopen, &gpclose, &rbread, &gpwrite, &vgsgtty, 0, 0,
         &dtopen, &dtclose, &dtread, &dtwrite, &fskbdtsgtty, 0, 0,
         &kbopen, &kbclose, &kbread, &kbwrite, &fskbdtsgtty, 0, 0,
         &fsopen, &fsclose, &fsread, &fswrite, &fskbdtsgtty, 0, 0,
         0       };




vgopen(dev,flag)
{ register int dminor;
  dminor = dev.d_minor;
  if (vgunit[dminor].status & OPEN) { /* its an open error */
        u.u_error = EIO;
        return;
  }
  vgunit[dminor].vg_procp = u.u_procp;
  (*vgdev[dminor].d_open)();
  VG_CONT->reg |=  intenable;   /* initialize and enable interrupts */
  vgunit[dminor].status |= OPEN;
}
```

```
vgread(dev)
{ register int dminor;
  spl0();
  dminor = dev.d_minor;
  (*vgdev[dminor].d_read)();
  VG_CONT->reg |= intenable;
}




vgwrite(dev)
{ register int dminor;
  spl0();
  dminor = dev.d_minor;
  (*vgdev[dminor].d_write)();
  VG_CONT->reg |= intenable;
}




vgclose(dev)
{ register int dminor;
  dminor = dev.d_minor;
  (*vgdev[dminor].d_close)();
  VG_CONT->reg |= intenable;
  vgunit[dminor].status = 0;
}
```

```
vgint(dev)
{ register int whichone;
  whichone = VG_STAT->reg >> 1;
  spl7();
  switch (whichone) {           /* find which device interrupted me */

   case GP_BUS:
        break;

   case DATABLET:
        dtintr();
        break;

  case KEYBOARD:
        kbintr();
        break;

  case FUNCTION:
        fsintr();
        break;

  case GPU_UNIT:
        gpintr();
        break;


  default:
        printf("VGERR - [%o]  : %o %o\n",whichone,PIN(stat),PIN(dcust));
        VG_CONT->reg = 020000;          /* clear the bad condition*/
  }
  VG_CONT->reg |= intenable|intack;
  spl0();
}
```

```
vgioctl(dev,command)
int     dev, command;
{
        (*vgdev[dev.d_minor].d_ioctl)(dev,command);
}

gpurestart()
{
  if (!(vgunit[0].status & RUNNING)) {
        if (vgunit[0].status & SLEEP){
                                /*
                                 * special case to speed
                                 * up GPU and system calls
                                 * should allow several vectors
                                 * to be drawn before refresh
                                 */
                wakeup(&vgunit[0].io);
                vgunit[0].status &= ~(RUNNING|WAITING);
        }
        POUT(cmd, (PIN(cmd) | GO | PICHE) & 077777);
        vgunit[0].status |= RUNNING;
        vgunit[0].status &= ~ WAITING;
  }
}

gpwait()
{
        if (vgunit[0].status & (RUNNING|WAITING)){
           vgunit[0].status |= SLEEP;
           sleep(&vgunit[0].io,PWAIT);
           vgunit[0].status &= ~SLEEP;
        }
}
```

Appendix E: Listing of File /sys/conf/makefile

The file /sys/conf/makefile is used to regenerate the system during execution of the command "make unix60".

Line 57 specifies the maximum allowable size (in bytes) of the system.

```
unix unix44 unix45 unix70:      l.o mch_id.o c.o ../sys/LIB1_id ../dev/LIB2_id
        @echo ""
        @echo "The output file will be named unix_id !!!!!"
        @echo ""
        ld -o unix_id -X -i l.o mch_id.o c.o ../sys/LIB1_id ../dev/LIB2_id
        @echo ""
        @echo "Size of unix_id is TEXT+DATA+BSS = TOTAL"
        @echo ""
        size unix_id
        rm *.o

all:    al140 al170

al144 al145 al170:
        cp ../h/param_id.h ../h/param.h
        cd ../sys; cc -c -O *.c; mklib_id; rm *.o
        cd ../dev; cc -c -O *.c; mklib_id; rm *.o

mch_id.o:       mch0.s mch_id.s
        as -o mch_id.o mch0.s mch_id.s

allsystems:
        @echo ""
        @echo "If not super user, this will not work !!!!"
        @echo ""
        mkconf <hptmconf
        make unix40
        mv unix_i /hptmunix_i
        mkconf <rptmconf
        make unix40
        mv unix_i /rptmunix_i
        mkconf <hktsconf
        make unix40
        mv unix_i /hktsunix_i
        mkconf <rltsconf
        make unix40
        mv unix_i /rltsunix_i
        mkconf <hphtconf
        make unix70
        mv unix_id /hphtunix_id
        mkconf <rptmconf
        make unix70
        mv unix_id /rptmunix_id
        mkconf <hktsconf
        make unix70
        mv unix_id /hktsunix_id
        mkconf <rltsconf
        make unix70
        mv unix_id /rltsunix_id

unix23 unix34 unix40 unix60:    l_i.o mch_i.o c_i.o ../sys/LIB1_i ../dev/LIB2_i
        @echo ""
        @echo "The output file will be named unix_i !!!!!"
        @echo ""
        ld -o unix_i -x  l_i.o mch_i.o c_i.o ../sys/LIB1_i ../dev/LIB2_i
        @echo ""
        @echo "If size of unix_i > 49152 bytes, UNIX IS TOO BIG !!!!!"
        @echo ""
        @echo "Size of unix_i is TEXT+DATA+BSS = TOTAL"
        @echo ""
        size unix_i
        rm *.o

mch_i.o:        mch0.s mch_i.s
        as -o mch_i.o mch0.s mch_i.s
```

```
al123 al134 al140 al160:
        cp ../h/param_i.h ../h/param.h
        cd ../sys ; cc -c -O *.c ; mklib_i ; rm *.o
        cd ../dev ; cc -c -O *.c ; mklib_i ; rm *.o

c_i.o:  c.c
        cp ../h/param_i.h ../h/param.h
        cc -c -O c.c
        mv c.o c_i.o
l_i.o:  l.s
        convert l.s l_i.s
        as -o l_i.o l_i.s

c.o:    c.c
        cp ../h/param_id.h ../h/param.h
        cc -c -O c.c
```

Appendix F: Creation of Special Files for the VG Graphics Device

Before

```
# ls /dev
console
kmem
lp
makefile
mem
mk_rk07b
mt0
mt1
nrmt0
nrmt1
null
r,t1
rmt0
rmt1
rp0
rp10
rp13
rp17
rp3
rrp0
rrp10
rrp13
rrp17
rrp3
swap
tty
tty00
tty01
tty02
tty03
tty04
tty05
tty06
tty07
tty08
tty09
tty10
tty11
tty12
tty13
tty14
tty15
vp0
```

Create Special Files

```
# cd /dev
# /etc/mknod gpu c 22 0
# /etc/mknod dtb c 22 1
# /etc/mknod kbd c 22 2
# /etc/mknod fss c 22 3
#
```

After

```
# ls /dev
console
dtb
fss
gpu
kbd
kmem
lp
makefile
mem
mk_rk07b
mt0
mt1
nrmt0
nrmt1
null
r,t1
rmt0
rmt1
rp0
rp10
rp13
rp17
rp3
rrp0
rrp10
rrp13
rrp17
rrp3
swap
tty
tty00
tty01
tty02
tty03
tty04
tty05
tty06
tty07
tty08
tty09
tty10
tty11
tty12
tty13
tty14
tty15
vp0
```

Appendix G: Major System Directories

```
# ls /sys/conf
c.c
c.c.lp
c.c.save
c.c.vg
conf.afit
conf.afit.lp
conf.asd
convert
dtb.c
fss.c
hkhtconf
hktmconf
hktsconf
hphtconf
hptmconf
hptsconf
kbd.c
l.s
l.s.auto
l.s.good
l.s.lp
l.s.save
l.s.vg
l_i.s
makefile
mch0.s
mch_i.s
mch_i.s.save
mch_id.s
mkconf
mkconf.c
mkdev_i
mkdev_id
mksys_i
mksys_id
rlhtconf
rltmconf
rltsconf
rphtconf
rptmconf
rptsconf
unix_i
unix_i.save
unix_id
unixconf
vg_conf.load
vg_conf.unload
vgtest.c
#
```

```
# ls /sys/dev
LIB2_i
LIB2_i.save
LIB2_i.vg
LIB2_id
bio.c
cat.c
dc.c
dh.c
dhdm.c
dhdm.c.orig
dhdm.c.v7m
dhfdm.c
dkleave.c
dn.c
dsort.c
du.c
dz.c
hk.c
hp.c
ht.c
kl.c
lp.c
mem.c
mklib_i
mklib_id
mx1.c
mx2.c
partab.c
pk0.c
pk1.c
pk2.c
pk3.c
rf.c
rk.c
rl.c
rl.c.orig
rp.c
rx2.c.v7m
sys.c
tc.c
tm.c
ts.c
ts.c.old
tty.c
vg.c
vp.c
vs.c
#
```

```
# ls /sys/h
acct.h
buf.h
callo.h
conf.h
dir.h
dumprestor.h
fblk.h
file.h
filsys.h
ino.h
inode.h
map.h
mount.h
mpx.h
mx.h
pack.h
param.h
param_i.h
param_i.h.v7m
param_id.h
pk.h
pk.p
prim.h
proc.h
pwd.h
reg.h
seg.h
smallparam.h
stat.h
stdio.h
systm.h
term.h
text.h
timeb.h
tty.h
types.h
user.h
vg.h
#
```

```
# ls /sys/sys
LIB1_i
LIB1_id
acct.c
alloc.c
clock.c
fakemx.c
fio.c
iget.c
machdep.c
main.c
malloc.c
mklib_i
mklib_id
nami.c
pipe.c
prf.c
prim.c
rdwri.c
sig.c
slp.c
subr.c
sys1.c
sys2.c
sys3.c
sys4.c
sysent.c
text.c
trap.c
ureg.c
#
```

Appendix H: Rebooting the System from UNIX Object File /unix.vg

This appendix contains a listing of the commands executed to reboot the system from UNIX object file /unix.vg. This rebooting session was accomplished from the system console. When the session was begun the system was in multiuser mode with the system console logged in as the "root" user executing a monitoring loop.

In this example, all commands typed by the systems programmer are under scored.

```
type a control D
# who
root      console Dec 18 21:11
# kill -1 1
# # fsc
******************************* FSC *******************************
Fri Dec 18 21:11:49 EST 1981
Check root file system, drive 0
/dev/rrp0:
files    621 (r=551,d=26,b=8,c=36)
used    8652 (i=231,ii=8,iii=0,d=8405)
free     570
missing    0
/dev/rrp0:
     entries  link cnt
2       12       13
Check home file system, drive 0
/dev/rrp3:
files   1622 (r=1434,d=188,b=0,c=0
used   18378 (i=508,ii=5,iii=0,d=17860)
free   15460
missing    0
/dev/rrp3:
Check /usr file system, drive 1
/dev/rrp17:
files   5316 (r=5088,d=228,b=0,c=0)
used   50757 (i=1211,ii=37,iii=0,d=49472)
free     917
missing    0
/dev/rrp17:
Fri Dec 18 21:15:22 EST 1981
******************************* FSC *******************************
# sync
# sync
# boot
Boot
: hk(0,0)unix.vg
mem = 203968
# date 12180918
Fri Dec 18 09:18:00 EST 1981
```

Vita

Bradley Ray Stewart was born on 5 August 1955 in San Luis Obispo, California. Bradley graduated with academic honors from East Union High School, Manteca, California in 1973. He attended Brigham Young University from which he received a Bachelor of Science degree in Computer Technology in June 1980. Upon graduation, he received a commission in the USAF through the ROTC program. He then entered the School of Engineering, Air Force Institute of Technology, in June 1980.

Permanent address: 1026 Lewis Oak Road
Gridley, California 95948

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/MA/81D-6	2. GOVT ACCESSION NO. AD-A115 582	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A UNIX BASED DEVICE DRIVER FOR THE VECTOR GENERAL 3404 GRAPHICS DISPLAY SYSTEM		5. TYPE OF REPORT & PERIOD COVERED MS Thesis
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Bradley R. Stewart		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT/EN) Wright-Patterson AFB, Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE March, 1982
		13. NUMBER OF PAGES 213
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

15 APR 1982

[illegible]

Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB, OH 45433

18. SUPPLEMENTARY NOTES

Approved for public release; IAW AFR 190-17

~~F. C. LYNCH, Major, USAF~~
~~Director of Information~~

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

UNIX
Computer Graphics
Peripheral Device I/O
Device Driver
Device Handler
Vector General 3404
PDP11/60

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A device driver for the Vector General 3404 Graphics Display System was installed under the UNIX version seven operating system on a PDP11/60 computer. This was accomplished by modifying an existing device driver which was designed to run under version six of the UNIX operating system.

The major topics addressed in this report are the C programming language, peripheral device I/O processing under UNIX, the hardware interface between the PDP11/60 and the graphics display system, the graphics display system

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

itself, and the existing device driver software.

Structure charts were used to document the design of the UNIX peripheral device I/O processing software and the design of the device driver software. Modifications to the original device driver were easily accomplished due to the top-down modular design of the original software. UNIX provided a straight-forward interface for adding the device driver software to the system.

ILMED

8