To the Graduate Council:

I am submitting herewith a thesis written by James T. McCrae entitled "An Interface Between the Unix and CP/M Operating Systems." I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael Moshell, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Vice Chancellor
Graduate Studies and Research

AN INTERFACE BETWEEN THE UNIX AND CP/M OPERATING SYSTEMS

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

James T. McCrae

August 1981

ABSTRACT

The purpose of this work was to design and implement the software required to utilize a hardware interface between a PDP-11/34 mini-computer and an Imsai 8080 micro-computer. The interface was designed and built by Andrew Wang at The University of Tennessee (Wang, 1978). The computers used were located in the University's mini-lab in the South College building on the Knoxville campus.

The interface built by Wang was parallel and asynchronous. A pair of control and data registers on each computer afforded the data link. On each system, transmit and receive operations were done through this same register pair. This put some limitations on the software capabilities. Software to make use of this interface through the computers' operating systems had not been developed previous to the current work.

In order to produce the needed software, it was necessary to first understand the two operating systems involved, Unix on the PDP-11 and CP/M on the Imsai. Once the author had become familiar with both systems and the interface, the operating systems were modified to accommodate user programs which might use the capabilities of the cross-computer interface. Finally, a virtual floppy disk system was designed and implemented.

This system uses the interface to send floppy disk format data from CP/M to a Unix user program. The Unix program maintains a set of disk files which simulate the CP/M format disks. Use of this system is relatively transparent to the CP/M user. Such a system was suggested in Wang's write-up of the interface. The software was also designed to give future users the ability to create new applications for the interface with a minimum of trouble.

PREFACE

This paper attempts to describe changes made to two operating systems in order to implement a computer-to-computer interface. The interfacing of computers is still a young field. The importance of this area of computer technology to computer applications of the future is becoming more apparent. As distributed processing, computer networks, and intelligent industrial machinery, to name a few applications, become more prevalent, the need for an understanding of software such as this paper describes will become more important to the average programmer. This is reflected in the recent trend toward the study of operating systems in academic curricula (Coffman and Denning, 1973, p. xi).

A major portion of the time taken in completing this project was devoted to reading and understanding the operating system code for Unix and CP/M. Therefore, the first two sections are overviews of the relevant attributes of these systems. These overviews are by no means exhaustive analyses of the code, but are aimed at providing the reader with sufficient understanding of the systems to grasp the work entailed in the completion of the project. The remainder of the sections describe

the stages of software development. The appendices contain the actual code that was written.

TABLE OF CONTENTS

PAGE

CHAPTER I

OVERVIEW OF CP/M OPERATING SYSTEM

The CP/M operating system was designed by Digital Research for use with micro-computer systems utilizing an Intel 8080 or Zilog Z-80 microprocessor. CP/M is a general purpose operating system providing users with extensive facilities for program development and implementation.

The main task of CP/M, as with any operating system, is to provide the user with easy access to the particular system's resources. The operating system is closely linked to the disks used on a system and maintains a sophisticated file system. The bulk of the operating system activity is done by the memory resident code loaded at boot time. This resident code is logically divided into three distinct sections. These are the Central Command Processor (CCP), the Basic Disk Operating System (BDOS), and the Basic I/O System (BIOS). These three sections interact hierarchically, with CCP at the highest level, that is the level closest to the user, and BIOS at the lowest level. Each section will be discussed separately later.

At any given time, CP/M is "logged into" one of the system's disks. The currently logged system is the default source for all file references made by the user.

Files other than those on the logged disk can be referenced by preceding the file name with the drive name followed by a colon (':'). To change the currently logged disk, the name of the drive to be made current is entered at the console followed by a colon and a carriage return.

In addition to the three sections of CP/M mentioned earlier, there is a Transient Program Area (TPA) where files residing on a disk can be loaded and executed. There are separate areas in the TPA for non-resident system routines and for user programs.

CCP is concerned with accepting input from the console device and interacting with the user through the console. Commands to CCP are actually the names of disk files to be executed in the TPA. The file system assumes certain conventions in the names of disk files, and CP/M supported utility programs observe these conventions. For example, the CP/M assembler utility program, ASM, accepts an argument that is assumed to be the name of an assembly language source file. The extension .ASM is then appended to the input string and the referenced disk is searched for this file. ASM will not assemble a file with any other extension but .ASM, since it does not search for any other files but .ASM files. Similarly, only files with the extension .COM will be loaded into the TPA and executed. The .COM

extension is automatically appended to executable files by the loader utility program, LOAD, before the file is written on a disk.

When a command is entered to CCP, the current disk is scanned for a file with the extension .COM. If the file is found, it is loaded into the TPA and control is passed to the start of the executable file image. CCP also accepts five internal commands as input. These commands provide file manipulation services and status information on disk files.

The file activities which CCP provides to the user are carried out by BDOS and BIOS. These two sections of code are frequently referred to collectively as FDOS and can be accessed from a common entry point.

BDOS is concerned with maintaining and utilizing the file systems on the system's disks. File maintenance centers around the File Control Block associated with each file. The FCB for a file is stores in the disk's directory and loaded into a memory location within the operating system code when the file is accessed. The FCB contains the file name, its size, and fields used in performing I/O on the disk file.

BIOS contains all the routines which actually perform I/O in CP/M. Entry to the various device driver routines is attained by way of a jump table, a list of jump instructions to addresses inside the BIOS code.

These addresses are the start addresses of the driver routines. When a driver routine finishes execution, control is returned to the calling program. This means that user programs can access BIOS directly if the address and sequence of the jump table is known for the particular CP/M system. This address will vary on systems with different memory sizes.

Data transfers in the 8080 microprocessor are done through one of 256 I/O ports. These ports are device independent. Any control necessary to accommodate the characteristics of a particular device must be handled by the device driver in BIOS. Control is normally achieved by having one or more ports dedicated to control and status operations for the device. Data is transferred through a separate port. A simple device, such as a console, needs only one control and status port, while complicated devices may need several control and status ports. For example, a disk controller requires five ports, one each for 1) output of the hardware command to the controller and input of the controller status, 2) output of the requested track number, 3) output of the requested sector number, 4) input and output of data, and 5) input of the wait status of the controller during hardware operations and output of control signals to the controller. Ports 1) and 5) above are bidirectional, as is the data port.

The early 8080 microprocessors, including the processor in the Imsai system used in this project, do not possess interrupt capabilities. Therefore, the BIOS driver routines must continually monitor the status of a device port to determine when an I/O transfer has completed or can begin. Some devices, specifically disk drives, are sufficiently fast that a short delay period can be used to wait for status signals from the device. These delays must be implemented in software of course. Most peripherals require that the driver routine sit in a loop, testing some status condition until the desired value is returned from the device. This approach would not work in a time-sharing operating system, since the CPU would be tied up continually waiting for I/O completion, depriving other executing tasks of CPU service. However CP/M is a single user system and the CPU is dedicated to a single executing task.

BIOS is designed for easy modification. Although the code must be modified in assembler language, reassembled, and patched into the rest of CP/M with utility programs, the use of a jump table in accessing driver routines demands that particular drivers reside in distinct sections of code. This provides a certain amount of modularity and localization to BIOS as a whole, so new drivers can be inserted into

BIOS with little danger of affecting the operation of existing routines.

CHAPTER II

OVERVIEW OF THE UNIX OPERATING SYSTEM

The Unix operating system was developed by Bell Laboratories for use with DEC's PDP-11 series of minicomputers. Unix incorporates some unusual concepts which separate it from most available operating systems. Although the main tasks of the operating system are no different from other systems, i.e., file management, CPU management, and I/O coordination, several features used to implement these tasks are unique to Unix.

Almost all of Unix is written in the C programming language, also developed by Bell Laboratories. A few of the lower level operations within Unix are written in assembler language due to the dependence of these operations on the particular machine being used. This part of Unix is different for the PDP-11/40 and below series and the PDP-11/45 and above series. Access to the assembler routines is transparent in the rest of the Unix code due to the C language's ability to directly reference assembled location names just like C function names. The function is the basic C unit of executable code, to be discussed later.

The C language affords the structured facilities of high-level languages such as PL/1 or Pascal while

allowing access to machine level activities in the manner of assembly language. The C compiler gives the programmer an unusual degree of freedom in manipulating variables which reference memory. This makes C a powerful tool for operating system construction since addresses can be directly accessed and manipulated within a framework of high level mnemonics and structured programming techniques.

The heart of Unix is the file system. The entire file system can be viewed as a rooted tree, with directories as non-terminal nodes and non-directory files as leaves. At the base of the tree is a directory called the "root." The root contains the addresses of more directories, from which every file in the system can be accessed. Any file in a directory may itself be a directory. Non-directory files, or text files, are all treated the same; no format or size restrictions are placed on these files except the availability of space on a particular device. One group of nondirectory files (the special files), does have special attributes, which will be discussed later.

To access a given file in the tree, it is only necessary to name all the directories in the path from the root to the file. If the current directory, that directory which the user is currently using, is in the path, the directories from the root to the current

directory may be omitted from the reference. The directory names in this path are called the "pathname" of the file. A pathname is formed by concatenating the directory names with slashes ('/'), in descending order left to right. The root is referenced by a single slash, so any file referenced from the root has a pathname preceded by a slash. For example, given a file named "text" in directory A, A in directory B, B in the root, the pathname for text would be /B/A/text.

Several important files reside off the root directory. One of these files is the "unix" file containing the operating system code which is loaded and executed at boot time. Several directories in the root contain frequently referenced system files, some for internal system use only and other services. One very important file in the root is the directory "/mount." The root directory resides on the system device, also called the root device. Any file not on this device must be referenced through /mount. This directory contains entries for all the currently mounted devices. Some of these devices are mounted at boot time; others may be mounted by the user via the "mount" system call. Each entry in /mount points to the root directory for that device. Once a device is mounted, the root directory and all directories below it fit into the tree structure like any other directory descending from the root.

Probably the most unusual feature of Unix is its approach to I/O operations. All I/O appears at the higher levels of Unix and to the user as being done on a file, regardless of the actual device being accessed. A read or write on a disk file in a user's directory space is done with the same system call as a read or write on a teletype or a tape drive. This is accomplished by distinguishing between standard files, those containing textual or directory data, and special files, which are associated with peripheral devices. Special files are entered into a directory off the root called "/dev." There is at least one entry in this directory for each peripheral device, including the root device. Special files are indicated by a unique bit pattern in the directory entry.

Associated with each special file are a pair of integers uniquely identifying the device. These are the major and minor device numbers. The major device number may be thought of as identifying a particular class of devices. For example, all RK05 disks on a given system may have the same major device number, while all RL01 disk drives would have another major device number. The minor device number uniquely identifies a single device. On a system with four RK05 disks and three RL01 disks we might have a major device number of 2 for the RL01's. The individual RK05's would then have minor

device numbers from 0 to 3 while the individual RL01 drives would have minor device numbers from 0 to 2. A major/minor device number pair of 1/2 would uniquely identify the third RK05 drive, while 2/0 would uniquely identify the first RL01 drive.

Devices are further divided into character devices and block devices, the difference again indicated in the "/dev" directory entry. Block devices generally do I/O in blocks of 512 bytes while character devices transfer data one byte at a time.

Actual I/O in Unix is performed by sets of functions referred to as device drivers. A given device driver contains all the code needed to perform I/O on that device. The drivers incorporated into a particular Unix system will depend on the hardware configuration of that system. Two tables within Unix provide access to the device drivers associated with each major device number, one table for character devices and another for block devices. A given device can have separate drivers to perform both character and block I/O; disk drives will frequently have drivers for both modes of I/O.

Before explaining the mechanics of the I/O system further, it may be helpful to explain some properties of the C language. In C, the basic unit of program structure is the function. A function would probably be called a subroutine or a procedure in another language.

When a function name is encountered in a C program, the routine is executed and the results and returned values are effective from that point on in the program. C is very flexible about where a function name may appear; a function name can be passed as an argument, or a function can be executed within the test portion of an "if" or "while" clause. Functions can even be executed in the argument list of other functions, the returned values then serving as the actual arguments to the outer nested level function.

The address of a function, or a variable, can be taken by preceding the function or variable name with an ampersand ('&'). The value thus formed is the actual resident memory location associated with that mnemonic. Conversely, when a variable defined as an address, such as an array or pointer variable is preceded by an asterisk ('*') the value formed is the contents of that memory address. These two forms of addressing mnemonic values can be nested; specifically, a function name which appears preceded by an ampersand can be executed by preceding that reference by an asterisk. This means that a function can be executed without the function name actually appearing in the line of source code where it is to be executed. This can be achieved by setting a pointer variable to an address which contains a function name and then preceding the pointer name by an asterick and enclosing the entire reference

within parentheses. When this construction is encountered in the course of execution, the function named in the location pointed to by the pointer variable will be executed exactly as if the actual function name appeared in the source code line. For clarification, assume that a variable "fp" points to some address containing the value "&func," where func is the name of a properly defined function. Then the construction

```
(*fp)();
```

will cause func to be executed. The second set of parentheses are the argument list delineators and identify the construction as a function name rather than a variable name. The semicolon is the standard C expression terminator.

Further flexibility in accessing memory locations within C programs is attained by use of structures. A structure in C consists of a list of variable identifiers which can be addressed as a unit. Each identifier in the structure retains its position relative to the other members of the list, that position being dependent on the size of the variable identifiers preceding it in the structure. The structure name is concatenated with the member name by a period ('.') or a dash followed by a right arrow ("->"). The construction

```
struct meta {
        integer alpha;
        character beta;
        integer gamma;
            } table
```

defines a structure with three elements - alpha, beta, and gamma. "Table" is defined as the name of a structure which can be referenced with these member names. To reference a character which is located one integer length past the address of "table," the construction

```
table.beta
```

or

```
table -> beta
```

can be used. Structure references can be nested. Given another structure definition

```
struct metamore {
        integer theta;
        struct meta;
            } supertable;
```

the same element referenced above can be referenced as

```
supertable.meta.beta
```

or

```
supertable->meta.beta
```

or other constructions formed with the period and right arrow.

The capabilities of C described above give some idea of the programming flexibility available with C. It is possible to write code which is easily understandable with C, and it is equally possible to write extremely cryptic and convoluted code. (Unix contains both.) The purpose of this overview is to describe relevant features of Unix, so C will not be examined in greater depth here. The above discussion should

provide sufficient background for the understanding of Unix I/O.

As stated above, there exist two tables which provide access to all Unix device driver routines. The entries in these tables are addresses of driver functions. Unix maintains a complex set of structures which are used to reference various system tables. One of these structures is called the "i-node" structure. It is a template used to access the contents of a directory entry. Every active file of any kind at a given time has its own uniquely numbered i-node. The i-node itself is a system maintained table which contains all the information needed to do I/O associated with the file. When a file is not active, the i-node resides on the device of the file in a system table called the i-list. Each i-node is uniquely identified by an integer pair, the first being the major and minor device numbers concatenated and the second the number of the i-node within that device's i-list. This value resides in the directory entry for the file.

When an I/O request is made for a file, the major device number is extracted from the directory entry by means of a structure reference and used to access the appropriate driver function within the appropriate system driver table. The character device driver table contains five entries for each major device number.

These are the addresses of that device's open, close, read, write, and special service functions. The last function is used to set or retrieve status information on terminal devices. The block device driver table contains four entries for each major device number, the addresses of the device's open, close, and I/O strategy routines and the address of a table used to maintain buffer control. The strategy function performs all physical I/O for the device. The sets of table entries in each table are also referenced by a structure unique to the table. These tables and the associated structures are the only means of access to the device drivers from within Unix.

When I/O is requested on a file, a field in the file's i-node is checked to see if the file is a special file. If so, this field designates it as character or block I/O type. This information is used to get a pointer to the proper driver table. The function required for the I/O is then executed by constructing its name out of structure member references. The major device number serves as the member identifier of the correct set of drivers in the table, and the system call identifies the member of the sub-table containing the function address. Control passes immediately to the function whose name has been constructed and the I/O is carried out. The minor device number is

passed to the driver for whatever use the driver requires.

From the user point of view, I/O is always done in connection with a file descriptor which is returned after a valid open request. This file descriptor is a unique integer for each file opened by the user process. Unix accesses the i-node associated with the file by way of a process table unique to each user process. The table contains a list of open i-nodes for the process, saved in the same order as the file descriptors maintained by the user.

I/O for non-special files is done the same as for special files except that there is more overhead involved for non-special files. This is due to the need to calculate the block number of the file on a device. Block I/O is more complicated than character I/O in general, because of the code needed to maintain buffers.

Another feature of Unix which is relevant to this paper is the connection between processes and terminal devices. Processes, like files, can be viewed as a rooted tree, with the boot time process at the apex. At boot time, a single process sets up the system tables necessary for future processes and takes control of the system's resources. All other processes are formed as offspring of this system process by means of the "fork" operation.

When a process forks, a duplicate image of the original process is formed. An image can be thought of as all the elements in memory which give the executing process access to system resources. This would include such things as register values, table pointers and contents, and the state of various control variables. During time-sharing activities, the image of a swapped out process is what is written out to the swapping device. After a fork operation, control is passed to the new process. Associated with each process is a process identifier, or process id. This identifier will be zero for the child of a fork operation and non-zero for the parent of the fork. By examining the process id, a process can tell if it is the child or parent of a fork. This is important since the images of the two are identical after the fork occurs.

During system initialization, a process is created for each terminal or time-sharing device on the system. The terminal devices are entered in a file named /etc/ttys. The entries consist of three characters per terminal device. The first character places the terminal in a change-of-command chain in the event that the system console is not available and control must be passed to another terminal. The second character designates a table to be used in setting the initial status and characteristics of the device for I/O. This information

is put in a table by the program "gtty," the terminal initiation program executed by each terminal process after the process is created. The third character is the identifying character for an entry in the device directory,/dev. The corresponding entry in /dev has a name of the form "ttyx," where x is the same as the character in the /etc/ttys entry.

After the "gtty" program has set the terminal characteristics, it issues a read to the device. When a user enters a character string to the terminal, "gtty" checks the characters to determine whether the terminal is upper-case only, what the end-of-line character is for the device (carriage return or linefeed), and what the line-kill and erase characters should be. Then the "login" program is executed. When a user logs in successfully, control passes to the "shell," the user monitor program which provides execution and maintenance services to the user. Once the "login" program begins, the process associated with the terminal lives to serve time-sharing users until a reboot is done.

CHAPTER III

DESIGN CONSIDERATIONS FOR INTERFACE SOFTWARE

Having discussed both the Unix and CP/M operating systems, we are now ready to examine the software modifications made to these systems to accommodate the micro-computer interface.

A central concern in the design of any operating system software is to provide the user with as much service as possible with a minimum of difficulty to the user. The file structure and I/O mechanics of Unix are aimed at making access to all system resources as similar as possible from the user's point of view. CP/M is a less sophisticated monitor program and therefore less transparent to the user, but the goal of easy user accessibility to resources is still evident. The task of any operating system is largely to relieve the user of the burden of managing system resources, leaving the programmer free to develop applications programs without worrying about the low-level control of the particular system's peripheral devices.

In designing the device drivers on both operating systems, the major concern was with making access to the interface facilities as transparent as possible to the user. Several difficulties were encountered in the design process. A major constraint on the design was the

fact that the Imsai system has no interrupt capabilities. Therefore, request for services must always originate from the CP/M side of the interface. A related problem was the need for a process within Unix to service the requests from CP/M. As previously noted, every process in Unix must come from a fork of the original system process. These children are all created in association with entries in the "/etc/ttys" directory. Although the Unix initiatlization program assumes that these devices are terminals of some type, no check is made on the major device number, driver, or anything other than the "ttyx" format of the device name. Since all system reads are identical within user programs, including the initialization programs, login, and the shell, the actual device to which the process is connected can be any device which is properly accounted for in "/dev" and in either the character or block device driver tables.

Given the above restraints - only CP/M can request service and only terminal devices are associated with Unix user processes - it seemed quite convenient to make the CP/M interface appear as a terminal device during Unix process initialization. The physical characteristics of the interface used are unique, however, and could not be handled by any of the standard terminal interface drivers. Therefore, a new major

device number was assigned to the CP/M interface and entered into the device directory "/dev" as "ttym," the "m" seeming appropriate for a micro-computer interface. The driver, named MC in accordance with DEC and Unix prejudice toward two character driver names, was designated as a character device driver and the names of the driver functions were entered into the character device driver table. The fact that the CP/M interface is not a terminal device at all is of no consequence to Unix, since all I/O operations are controlled by the MC driver functions.

The major consideration in designing the MC driver functions was determining the nature of the data to be passed between the two systems. The necessity of attaching the interface to a terminal user process had definite constraining implications. The standard terminal drivers in Unix all use a set of higher level functions to perform modifications on the terminal input. Many characters are interpreted as escape characters and are not directly transferred to the user process. This is done by placing each line of input from a terminal device on a raw character list until a new-line character is encountered. After a full line is entered, the raw list is transferred into a canonical list in which all special character interpretations and substitutions have been completed.

This new list is referred to as "cooked" input. The cooked input is then passed to the user process. The standard Unix user service programs and the shell expect all input to be in cooked mode and act on it accordingly.

One application of the interface would be to use CP/M as simply another time-sharing device. In this case, the standard Unix terminal functions could be used to interpret input lines from CP/M for use by the shell and user programs. Accommodating the interface driver functions to the higher level functions which turn raw input into cooked input would be an easy enough task. However, the immediate application desired was for a virtual disk system serving CP/M. This requires a different approach to input, since a CP/M user must be able to write and read any character on the virtual disk without modification or interpretation. Furthermore, it would seem that restricting a micro-computer system with CP/M's capabilities to serve as a non-intelligent terminal to Unix would be a short-sighted waste of resources. In short, having an entire computer system serve as a terminal to another computer system defeats the purpose of a computer-to-computer interface.

After considering the above set of circumstances, two alternatives emerged which would allow the interface to serve as both a timesharing terminal to Unix and as

a true data link with interpretation of the data left to user programs on either side of the interface. One alternative involves use of the Unix "stty" command, available to users. This command allows the user to set the mode of input transmission from cooked mode to raw mode. After setting a terminal to raw, input is passed to the user process as it is entered at the device, without modification. To use this option, CP/M would first have to send the command "stty raw" followed by a carriage return to Unix. Then data transfers could proceed in raw mode until CP/M sent the command "stty -raw" followed by a carriage return. Then input transmission to the Unix user process would revert to cooked mode.

Consideration of this alternative raised an important point. Since CP/M is actually a user of Unix, at some point a CP/M program must login to Unix and pass the name of a desired application program or request some monitor service from the shell. These commands to Unix could either come from inside a CP/M user program which contained the command strings in ASCII, or they could be entered at the CP/M console, interpreted by a user program, and then sent to Unix. Most of the interpretation done by the Unix terminal functions in turning raw input to cooked input is for the benefit of the terminal user and of little consequence to the user process. When a user is entering input lines from a terminal it is

advantageous to have back-space and line-kill characters to correct mistakes made at the terminal keyboard. But given the facilities available to the CP/M user, it would seem more efficient to have the majority, if not all, of the commands sent to Unix originate in CP/M user programs. Once the CP/M program is written there would be no need for Unix to scan the input list for error-correcting characters. Furthermore, it would be a simple task to write a CP/M user program which interprets the CP/M console input and sends it to Unix in cooked mode, allowing the CP/M user to use Unix as any other time-sharing user.

It was decided that this second alternative would give the CP/M user maximum flexibility in accessing Unix resources. The user could utilize the CP/M terminal simulator program to login to Unix as a standard user and develop a desired application program to handle whatever input was to be transferred. Then the program could be run either by passing the program name via the terminal simulator or by including transfer of the program name at the beginning of a CP/M application program which interacted with the Unix application program. Once the programs had begun execution on both sides of the interface, all data transferred would be in raw mode and the programs would be free to interpret the data in any way desired.

The decision was made to keep the Unix MC driver code as simple as possible. Specifically, it was decided that the driver should perform no interpretation on the input from CP/M. Although the first alternative stated above, that which utilized the Unix "stty" command, would achieve the same ends as the second alternative, the second approach seemed more justified in terms of the overhead to Unix. The majority of the data transferred across the interface would most probably be associated with applications programs and not data sent for the purpose of time-sharing under Unix. This approach places the burden of data interpretation during time-sharing service on the CP/M system. CP/M is a single user system in which time constraints are of significance to only one user. Unix is a time-sharing system which must service several users simultaneously, and the added overhead of passing all CP/M input through an extra set of functions, although small, would affect all the users on the system.

CHAPTER IV

DIAGNOSTICS AND SOFTWARE TESTING

After the initial design of the software on both sides of the interface had been completed, several stages of diagnostics and testing were carried out. The initial diagnostics were aimed at verifying proper functioning of the interface hardware. These tests were made while the PDP-11 was down due to a disk problem. Two simple CP/M programs were written, one to test the transmission functions and one to test the receiving functions of the interface. Control of the interface from the PDP-11 was done from the operator console. For the Imsai-to-PDP transmission test, a CP/M program generated ASCII characters one at a time and sent them to the PDP-11. The values received on the PDP-11 Unibus were then checked by entering the control register and data register addresses into the console address register and examining their contents. This was done for the entire character set. All values transmitted were correct. For the PDP-to-Imsai transmission test, a CP/M program was written to read one character at a time from the interface and display the character on the CP/M console. Data was passed from the PDP-11 by entering the data into the interface data register from the operator console and then

entering the value into the interface control register to set the interface "run" bit. This test was expedited by first loading the lowest octal value which represents a printable ASCII character into the interface data register and then incrementing the data register to generate successive characters. This test also demonstrated correct transmission of all characters sent.

During the above described testing phase, it was found that the "error" status bit on the PDP-11 control register was being set after two consecutive Imsai-to-PDP data transmissions. This finding had two implications. First, the error bit could not be utilized by the Unix driver software to reliably detect errors. Second, referring to Wang's documentation of the interface wiring, it was observed that the interrupt vectored to location 154 would be generated when the error bit and "enable interrupt" bits were both set. This meant that the 154 interrupt facility could not be used.

After demonstrating that the interface was capable of reliable data transmission, the next step was to create a version of Unix which would allow access to and control of the interface for software testing purposes. This entailed modification of a number of Unix system files. First, it was necessary to place the names of the interrupt handler functions in the locations to which the interface interrupts had been vectored through the M1710

board. Unix sets these vectors at boot time by loading a file called "low.s" into the low address segment of memory. This file is an assembly language file. Introduction of the new vectors was done by setting the location counter to the vector address with the PDP-11".=(address)" facility. Then the address of another location defined in the file was inserted, followed by the processor priority to be loaded upon occurrence of the interrupt. The location pointed to by the address at the vector location contains a call to the actual interrupt handler. The "low.s" file was modified to transfer control to functions "mci150" and "mci154" upon interrupts vectored to locations 150 and 154 respectively. The interrupt vectored to location 150 is an asynchronous interrupt, controlled solely by handshaking operations. The interrupt vectored to location 154 was designed to occur whenever the "interface done" bit was set in the interface control register. However, this interrupt was rendered unusable, as discussed previously.

The next step in creating a test Unix system was to provide a separate set of files for use at boot time. This was accomplished by creating a new directory, "/mc," with modified versions of the files "init" and "ttys." Normally, these files are referenced through the directory "/etc." A new "ttys" file had to be created with the entry for device "ttym." Since this device is not accounted

for in the present Unix installation it was necessary to provide access to the new file for the test system only. The "ttys" file is read by the "init" program, so it was necessary to provide a separate version of "init" in the "/mc" directory which would reference the new "ttys" file. To complete the installation of the test system, it was necessary to change the "main" system function, which is executed as Unix is being initialized. "main" executes the "init" program to start the system processes. The name of the "init" program to be run is hidden in a table of integers in "main." The integers are actually a string of ASCII characters represented in integer format. The modification entailed encoding of the new character string, "/mc/init," into integer format and inserting it in the table in place of the "/etc/init" reference.

The changes to "init" and "main" would be temporary since a production environment incorporating the interface would reference the standard "/etc/tty" files with the new terminal entry.

The next stage of creating a Unix test system was to make a new bootable file in the root directory. This file was named "mcunix." An archive file is a group of files which can be maintained and manipulated as a single file. Unix system source files are kept in two archive files. A test MC driver was written and inserted in the

Unix source archive file called "drivers." One of the archive files is called "system" and contains those system files which generally do not change from one installation to the next. The program "main" is in this file. The other archive file is called "drivers" and contains the files which are specific to the hardware configuration of an installation. To add a new driver, the driver source file is simply archived into the "drivers" archive file. To generate a new Unix system, the two archive files are compiled and loaded along with the assembler files "low.s" and "m40.s" or "m45.s," depending on the PDP-11 being used. "m40.s" is used for series PDP-11/40 and below while "m45.s" is used for series PDP-11/45 and above. Another file, "config.c," contains the character and block device driver tables discussed in the Unix overview section. This file is compiled and loaded along with the others to create a new bootable Unix file.

A test driver was written and inserted in the "mcunix" bootable file. To test the driver, simple user programs were written for CP/M and Unix. The CP/M program accepts characters from the console and sends them to Unix until a rubout character is encountered in the input, at which time a null character (octal zero) is sent to Unix. The Unix program reads files from the interface and writes them into a file. When a null character is input, the Unix

program exits. Initially, the Unix login program is waiting for a read system call to return. Therefore, the first characters entered to the CP/M program must be a valid user name and password, if required. This logs the CP/M user onto Unix. Then the name of the desired program is entered, which the Unix shell then executes. After the program starts running, test data can be entered to the CP/M console.

The running of these test programs revealed a problem with the interrupt functioning of the interface. The first time the interrupt bit was set by the CP/M program, the interrupt was generated on the PDP-11 unibus as expected. However, subsequent attempts to generate the interrupt failed. Further testing revealed that by resetting the Imsai system from the console and rebooting Unix, the interrupt capability was restored but for only a single interrupt.

Referring to the documentation produced by Wang on the interface hardware, it was observed that if the 154 vectored interrupt is selected for transmission on the M1710 board, the 150 vectored interrupt cannot be generated, due to the wiring of the circuit. The 154 interrupt select signal can be generated by the "error" signal, which was known to be spuriously set after one data transmission, but the 154 interrupt select signal also required that the "interrupt enable" bit be set on the

interface control register. This was not being done by the Unix driver software. However, the coincidence of these findings strongly suggested that the malfunction in the error signal was connected to the once-only interrupt capability of the unibus.

Examination of the test routines used by Wang to verify proper functioning of the interface hardware revealed that this condition would not have been revealed by the tests. The interrupt vectored at 150 was used once in each test. Subsequent data transfers were controlled by testing the done status bits on both the PDP-11 and the Imsai within program loops. This was a feasible approach under the test circumstances, since stand-alone assembler language routines were used on the PDP-11 end of the test, and time-sharing constraints were not involved.

These findings cast sufficient doubt on the reliability of the interrupt generation circuitry to abandon the use of interrupts in the software. Although this development was a temporary setback, it could not be considered sufficient cause to abandon the project. In production environments, it is not at all unusual to discover that existing hardware is unsatisfactory for the required application, and, given time and financial constraints, the programmer assigned to completion of the application may very well be forced to make the best of the situation

at hand. If replacement of malfunctioning hardware is not feasible, the programmer must find a way around the problem.

Fortunately, the facilities within Unix afforded another possible approach. This alternative was to treat the interface as a handshaking device with no interrupt capabilities. The implementation of this approach on single user systems such as CP/M, which must run on hardware which lacks interrupt capabilities, would not be acceptable on Unix. CP/M, as discussed earlier, relies on loops to monitor the status of peripheral devices. This requires continuous control of the CPU until the loop terminates. Unix is a time-sharing system which must be able to service several users simultaneously, and allowing system device drivers to control the CPU in this manner would deprive users of access to the CPU, essentially locking up the entire system except for one device driver.

Unix provides two system facilities which make the no-interrupt handshaking approach to the interface software possible. These are the "sleep" and "wakeup" system functions, which are used together, and the "timeout" system function. The "sleep" function enables execution of a process to be suspended in association with an arbitrary memory location, provided as an argument to the function. The process id of the suspended process is placed on a list associated with that memory location,

along with a priority, also passed as an argument. The "wakeup" function is passed a memory location as an argument. All processes in the list connected to the location are scheduled for execution at the requested priority, with execution proceeding from the point immediately after the last "sleep" request. These two system calls are extensively used in connection with interrupt driven devices. A high level routine usually goes to sleep on the head of a character list used to accumulate input or output data for the device. When an interrupt occurs, the lower level interrupt handler wakes up the higher level process, usually after the character list has been filled or emptied to some predefined limit. The high level routine then handles the passing of data up to the user process. This was the scheme originally used for both input and output handling in the interface driver code.

The "timeout" system function allows a process to schedule a function to be run after some time interval has elapsed. The time interval is passed as an argument. Execution of the function requesting the timeout continues.

The handshaking operation was implemented by having the low level character read and write routines schedule a function after signaling the beginning of a data transfer. The character transfer functions then go to sleep, awaiting completion of the data transfer. The function which

was scheduled with the timeout begins execution after an interval which should allow sufficient time for the data transfer to complete. This function wakes up the sleeping character transfer function, which then tests the status of the interface to see if the transfer is complete. If it is, control is passed to the higher level read or write routine for character manipulation in conjunction with the user program. If the transfer has not completed, the sleep/wakeup cycle is repeated until completion. In the case that the Imsai stops sending data, this cycle continues indefinitely with no harm to the rest of the Unix system, outside of the added overhead of rescheduling the functions from time to time.

Although this implementation incorporates more overhead than the interrupt approach, due to the constant requests for Unix system process manipulation, it was found to work acceptably with the test programs. Determining the exact period of time required to complete a data transfer would take a considerable amount of testing, but if this interval could be narrowed down to the approximate time elapsed in a single transfer, the user would be serviced as efficiently as any other time-sharing user on a high speed terminal device. Other time-sharing users are not disproportionately slowed down by the operation of this software because all the functions involved are

scheduled for execution and do not simply grab the CPU from an already executing process.

CHAPTER V

MODIFICATIONS TO CP/M SOFTWARE

It was decided to provide the CP/M user with two modes of access to the Unix interface. The user can access the interface either directly by calling the single character transfer routine in BIOS or by referencing one of the non-implemented disk drives on the particular CP/M system. The latter access mode allows the user to treat the interface as a disk device transferring 128 bytes of data at a time. The user requests service from CP/M in this mode exactly the same as if a disk was being accessed. In this mode, the user has all the facilities of CCP, BDOS and BIOS at his/her disposal. This code is intended for use with the virtual floppy system so certain preliminary setup programs required for that system to work must be executed if results are expected. Of course, a user can develop his/her own applications software on the Unix system to make whatever use is desired of the data passed. This would require familiarity with a few conventions, to be discussed later.

The code needed for character transmission across the interface is relatively simple. Since the interface is parallel and asynchronous, the device routine only needs to place the data byte in the data port and then wait for completion of the transfer. This entails testing a bit

in the status port until it is set by the software at the other end of the interface. To receive data, the same bit is tested until it is set, at which time the data byte will be in the data port and can be retrieved.

The incorporation of the interface software into the BIOS section of CP/M entailed several modifications. Access to the various device drivers in BIOS is done via a jump table, as previously discussed. When a request is made for disk I/O, a jump is made to either the disk read or write code in BIOS. In order to provide access to the interface, this jump table was modified so that the jump is made to a section of code which tests the disk drive being used for the I/O request. The name of the current disk drive is stored in a common byte within BIOS and is set by other BIOS routines. If the disk being used is an implemented drive, either A or B on the Imsai system, a jump is made to the standard BIOS disk routine. If the drive is other than A or B, a jump is made to a separate routine which uses the interface for data transfer. Since the routine chosen for I/O is dependent only on the current disk, the user need only reference a non-existent disk in a standard command string in order to access the interface. No further action is necessary.

Using the interface in this way implies transfer of 128 bytes of data, the number of bytes in a CP/M format disk sector. This section of code was designed specifically

to be used with the Unix virtual disk simulator program, discussed in a later section.

The interface driver code in BIOS is divided into two levels. When the virtual disk facility is accessed, control passes to the higher level which performs tasks required for the virtual disk system. This entails first passing a character to specify whether a read or write operation is required. The program on the PDP-11 side of the interface must know the meaning of the character passed. As currently implemented, a +1 is sent to indicate a read request and a -1 is sent for a write request. After the I/O operation has been specified, the currently selected track and sector are sent to the PDP-11. Both of these values are set and stored from within BIOS and are assumed to be realistic values, being normally set by requests from BDOS. After the command code, track, and sector have been set, the driver waits for a return code character to be sent from the PDP-11. If this character is non-zero, the routine returns to the caller with the non-zero condition code. This is for the benefit of BDOS, which interprets a non-zero return code as indicating an error condition. The interpretation placed on the perceived error will depend on what BDOS was trying to do; BDOS does not always correctly identify the cause of error conditions. If the value returned by the PDP-11 program is zero, the driver proceeds to send or receive 128 bytes across the interface. The

bytes are taken from or moved into the buffer currently pointed to by the common BIOS buffer address word DMAADD. This will usually be the standard BIOS buffer but can be any location in memory. When the 128 byte transfer is complete, control is returned to the calling program.

Obviously, the code executed by accessing the interface via the BIOS jump table is very specialized and not suited to uses other than the virtual disk system. Any operating system on the PDP-11 which possesses a functioning driver can use the facilities of this system. However, it is possible to access the interface without going through the section I/O routine described above. The section of code which performs the interface data transfers is very simple. The location pointed to by the H-L register pair is used as the source or destination of a single transfer character, and then control is returned to the calling routine. Any user can use either the character at a time read or write routines, provided the user knows the address of the required routine. No check is made on the value of the H-L register pair or the data byte transferred. By accessing these low level routines, a user is totally unrestricted in using the interface, making it available to a much wider range of applications than merely the virtual disk system. This is an important asset to both the PDP-11 and Imsai systems, allowing more advanced usage such as implementation of shared processing or distributed

processing facilities. The drawback is that the user must know the addresses of the required routines, which may vary if modifications are made to BIOS or if the memory size of the CP/M system is altered.

CHAPTER VI

UNIX INTERFACE DRIVER

By making use of existing Unix routines, it was possible to reduce the interface driver code to a minimum. Since no interpretation of data was to be done, the code could be very direct and clear. When a read or write system call is made in Unix, control is passed to a set of functions which check the validity of the file descriptor used, locate the proper i-node for the file, and set an internal pointer in the processes' user structure to the buffer address passed as an argument in the call. Control is then passed to the driver associated with the accessed file. The driver which is called is passed the address of the user structure. The user structure is a control block maintained for the user by Unix. It contains the address of the buffer to be used for I/O and the number of bytes to be transferred before the driver returns. (The user structure contains many other fields and pointers which do not concern us here.) The system functions "cpass" and "passc" take care of byte transfer count and buffer maintenance for write and read operations, respectively. This is done through the user structure itself and requires no other code.

The driver for the interface contains four high level functions, read, write, open, and close. The open function is only needed to set a flag which the write function uses to determine if the interface is being used for write operations. The login and shell programs send an output string to the terminal device requesting user input, "login:" from the login program and "%" from the shell. Since the interface has only one control register and one data register, only one operation at a time can be using the interface. Therefore, the driver must be waiting for input data at the time a CP/M user logs on to Unix. Otherwise, CP/M would be trying to send a character while Unix was trying to send a character and neither side would detect the other's input. For this reason, the write function only writes data after a Unix user program has opened the interface. When a user is done with the interface, a close must be issued, which clears the bit indicating that character output goes to the interface. After a close and before an open, all output from Unix is simply ignored.

The read function uses a lower level function called getmc to pick up characters from the interface. In the original design, this was done by sleeping on the status word until an interrupt occurred. In the final working version of the driver, the getmc function is called at even intervals. If getmc detects that a character has

been sent, it is passed up to the main read function, which then uses the passc system routine to place the character in the user's buffer area. The read routine continues to request characters from getmc until passc returns a negative value, indicating that the user has all the characters requested in the read call. The write function operates the same way and uses the same timeout/wakeup function for timing purposes. Again, the write function continues to send characters to the putmc character transfer function until cpass returns a negative value indicating the write call is completed.

This driver uses very simple logic and entails a minimum of overhead under the circumstances. The only problem encountered during testing was that of logging out. The Unix routines which produce cooked output for the user also detect the end-of-transfer character(octal 4) which prompts the shell to log a user out. This character was not properly handled by the shell with this driver, since raw input of the end-of-transfer character to the shell is not interpreted as a request to log out. The problem was circumvented by having the CP/M logout routine send the command "login" with no argument. This causes the shell to log the user out and request a new login name.

CHAPTER VII

THE UNIX VIRTUAL DISK PROGRAM

As discussed previously, CP/M was modified to accommodate a virtual disk program on the PDP-11 side of the interface. The program "virtflop" performs the virtual disk simulation operations. Use of this program requires that the CP/M user observe several conventions. A set of CP/M user programs were written to facilitate use of the program.

When a CP/M user wants to use the virtual disk facility, he/she must first run the "LOG11" CP/M program. This program takes a user name and password and sends them to Unix. Presumably "login" is requesting input from the interface at that time, so the characters sent will go to the login program and the user will be logged into Unix. Then the user must establish contact with the Unix virtual disk program. If the user has no virtual disks on Unix or desires to create a new one, the CP/M program "CREAT11" is run. This program sends the name of the Unix virtual disk creation program, "Makedisk," to the Unix shell. Then "CREAT11" accepts a character string from the user which is sent to "makedisk" and used to create two files needed for the virtual disk. This is all that is necessary to define the disks under the Unix program. When the CP/M

user wants to use an existing virtual disk, he/she must run the CP/M program "MOUNT11." This program sends the name of the Unix virtual disk maintenance program, "virtflop," to the Unix shell. "virtflop" then continues to execute and perform data transfers until the CP/M user runs the program "UMOUNT11" which sends a command to "virtflop" indicating that the interface service is no longer needed. "virtflop" then exits and Unix control returns to the shell.

As with the "makedisk" program, the CP/M program MOUNT11 accepts a character string from the user which is sent to "virtflop." This string is used as the prefix for two files and uniquely designates which virtual disk the user wants to use. The two files are named xxxxxxxxxx.ind and xxxxxxxxxx.dat, the x's being the string of up to ten characters sent from CP/M. If these files do not exist, an error condition is returned to CP/M (actually to BDOS, as described earlier) and virtflop exits. If they do exist, virtflop returns a zero indicating all is well, and then enters an infinite loop within which the disk service is continued until an exit request is sent from CP/M.

The two files used by virtflop are named by concatenating the extensions ".ind" and ".dat" onto the string received. The first of these files, the ".ind" file, is an index into the second file, the ".dat" file. The index file consists of a sequence of track/sector pairs received from the BIOS virtual disk routines. These pairs uniquely

identify the sectors of the virtual disk, which are stored in the data file, ".dat." Indexing is done by determining an offset into the data file. This offset is the number of track/sector pairs scanned until the requested pair is found. This offset is then multiplied by 128 and used as an offset into the data file. The data file has no format. The only connection between the 128 byte sectors and the index files is in the offset calculation. If a track/sector pair is sent by BIOS which does not appear in the index file, virtflop creates a new index entry in the case of a write operation. If a read operation is in progress, an error is returned. The sectors in the data file and the track/sector pairs in the index file are stored as they come from BIOS and have no other sequential order other than temporal. This simple approach minimizes the time spent in accessing the files and serves the purpose.

CHAPTER VIII

SUMMARY

This project was predominantly a study in operating system modification. It was also a study in compromising theoretical design with physical limitations, a situation which arises more often than not in the world of computer applications. A system or applications programmer may not always be able to come up with an elegant solution to a problem. However, as a former co-worker of the author liked to say, there is a certain elegance to something that works. The goal of producing software which would give the user access to the interface facilities with a minimum of trouble was, I believe, attained.

The major problem involved in implementing this software is the shortcomings of the interface. It was found that powering down the Imsai with the interface connected caused Unix to trap with an unknown error condition. This trap condition can only be remedied by rebooting Unix. Because of the method Unix uses to update disk file structures, i.e., update information is saved in memory and written out to disk periodically, the file system is very vulnerable to such system crashes.

The problem of unexpected traps and the lack of interrupt capabilities on the existing interface could be

bypassed by using a standard terminal interface. Modification of the existing software to accommodate a different interface would not be too difficult. Specifically, only the low level Unix driver functions, "getmc" and "putmc," and the low level BIOS routines would need to be changed. The remainder of the software would be satisfactory.

To facilitate use of the CP/M software for single character transfers it would be nice to have an entry through BDOS which would provide access to the low level routines. This would make the interface more transparent in future applications.

In the virtual floppy disk system, it would be nice to incorporate a file compressing program to save disk space taken up by the virtual data files. Such a program could scan the directory tracks for a disk and remove those sectors associated with zeroed directory entries. As the software now stands, deleted files cannot be detected since CP/M simply zeroes the directory entry and does nothing to the actual data on the disk.

REFERENCES

REFERENCES

Coffman, Edward C., Jr. and Peter J. Denning. Operating Systems Theory. Englewood Cliffs, NJ: Prentice-Hall, 1973.

Digital Research. CP/M Interface Guide. Pacific Grove, CA: Digital Research, 1978.

Digital Research. CP/M System Alteration Guide. Pacific Grove, CA: Digital Research, 1978.

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

Ritchie, Dennis M. and Ken Thompson. The Unix Time-Sharing System. Murray Hill, NJ: Bell Laboratories, 1974.

Ritchie, Dennis M. and Ken Thompson. Unix Programmer's Manual. Sixth Edition. Murray Hill, NJ: Bell Telephone Laboratories, Inc., 1975.

Wang, Andrew. An Interface Between the PDP-11 Computer and the Imsai Microcomputer. Knoxville, TN: University of Tennessee, Knoxville, 1978.

APPENDIXES

APPENDIX 1

UNIX TO CP/M TRANSMISSION DIAGNOSIC

```
                        *** CRT IN/OUT TEST ***

0070 =          CSTAT   EQU     70H
0071 =          CDATA   EQU     71H
0001 =          CKBR    EQU     1
0002 =          CPTR    EQU     2

0100            ORG     100H

0100 CD0F01     LOOP:   CALL    UNIXIN

0103 FE7F               CPI     7FH         ; STOP ON ZERO
0105 CA0500             JZ      5           ; GO TO CCP
0108 4F         MOV     C,A                 ; SAVE CHAR

0109 CD3301             CALL    CONOT       ; ECHO CHARACTER

010C C30001             JMP     LOOP

                        *** ROUTINE TO GRAB A CHAR FROM UNIX ***

010F AF         UNIXIN: XRA     A           ; CLEAR ACCUM
0110 D303               OUT     3           ; SELECT STAT PORT
0112 DB01               IN      1           ; READ IT
0114 B7         ORA     A                   ; SET FLAGS
0115 F20F01             JP      UNIXIN      ; LOOP TIL UPPER BIT SET

0118 3E80               MVI     A,80H       ; SELECT DATA PORT
011A D303               OUT     3
011C DB01               IN      1           ; READ DATA

011E 4F         MOV     C,A                 ; SAVE CHAR
011F 3E80               MVI     A,80H       ; SEND 11 WAKE UP SIGNAL
0121 D303               OUT     3
0123 D301               OUT     1
0125 79         MOV     A,C
0126 C9         RET

0127 DB70       CONIN:  IN      CSTAT
0129 E601               ANI     CKBR
012B CA2701             JZ      CONIN

012E DB71               IN      CDATA
0130 E67F               ANI     7FH
0132 C9         RET

0133 DB70       CONOT:  IN      CSTAT
0135 E602               ANI     CPTR
0137 CA3301             JZ      CONOT
013A 79         MOV     A,C
013B D371               OUT     CDATA
013D C9         RET

013E            END     100H
```

APPENDIX 2

CP/M TO UNIX TRANSMISSION DIAGNOSTIC

```
                ***** DIAGNOSTIC CODE - CP/M TO UNIX *****
                *
0100                    ORG     100H
                *
                *
007F =          LASTX   EQU     7FH     ; 1ST UNPRINTABLE CHAR - [RUBOUT]
                *
0100 CD2E01             CALL    INTR11  ; GENERATE INTERRUPT
0103 CD2401             CALL    RDYWT   ; WAIT FOR TRANSFER COMPLETION

0106 0621               MVI     B,21H   ; START W/ 1ST PRINTABLE CHAR
0108 3E80       XLOOP:  MVI     A,80H
010A D303               OUT     3       ; SELECT DATA PORT
010C 78                 MOV     A,B     ; GET NEXT CHARACTER
010D FE7F               CPI     LASTX   ; IS IT PAST THE LAST CHAR ?
010F CA1B01             JZ      ENDIT   ; IF SO, SEND A 0 AND QUIT
0112 D301               OUT     1       ; OTHERWISE, SEND CHAR TO UNIX
                *       CALL    INTR11  ; GENERATE INTERRUPT
0114 CD2401             CALL    RDYWT   ; WAIT FOR TRANSFER COMPLETION
0117 04                 INR     B       ; SYNTHESIZE NEXT CHAR
0118 C30801             JMP     XLOOP   ; BACK JTO TRANSFER LOOP

011B AF         ENDIT:  XRA     A       ; WE'RE DONE. A <- 0
011C D301               OUT     1       ; SEND IT TO SIGNAL FINIS
011E CD2401             CALL    RDYWT
0121 C30500             JMP     5


0124 AF         RDYWT:  XRA     A       ; CLEAR A TO
0125 D303               OUT     3       ; SELECT CONTROL PORT
0127 DB01               IN      1       ; READ IT
0129 B7                 ORA     A       ; SET FLAGS
012A F22401             JMP     RDYWT   ; & TEST FOR DONE BIT HI

012D C9                 RET             ; WHEN IT'S HI, BACK TO CALLER

012E AF         INTR11: XRA     A       ; GET ZERO IN ACCUM
012F D303               OUT     3       ;   TO SELECT CSR PORT
0131 3E04               MVI     A,4     ; SET BIT 6 ON 11 CSR HI
0133 D301               OUT     1       ; TO GENERATE INTERRUPT
0135 C9                 RET             ; BACK TO CALLER

0136                    END     100H
```

APPENDIX 3

CP/M COMMAND TRANSFER PROGRAM

```
                        *****   DIAGNOSTIC CODE - CP/M TO UNIX  *****
                        *
0100                            ORG     100H
0070 =                  CSTAT   EQU     70H
0071 =                  CDATA   EQU     71H
0001 =                  CKBR    EQU     1
0002 =                  CPTR    EQU     2
                        *
                        *
007F =                  LASTX   EQU     7FH     ; 1ST UNPRINTABLE CHAR - [RUBOUT]
                        *
                        *       CALL    INTR11  ; GENERATE INTERRUPT
                        *       CALL    RDYWT   ; WAIT FOR TRANSFER COMPLETION
                        *
0100 CD4D01             XLOOP:  CALL    CONIN
0103 4F                         MOV     C,A
0104 CD5901                     CALL    CONOT
0107 3E80                       MVI     A,80H
0109 D303                       OUT     3       ; SELECT DATA PORT
010B 79                         MOV     A,C     ; GET NEXT CHARACTER
010C FE7F                       CPI     LASTX   ; LAST TRANSFER ?
010E CA0500                     JZ      5       ; IF SO BACK TO MONITOR
0111 CD3B01                     CALL    UTOL    ; CONVERT UPPER TO LOWER CASE
0114 CD4401                     CALL    CRLF    ; CHANGE CR TO LF
0117 D301                       OUT     1       ; SEND CHAR TO UNIX
0119 CD2901                     CALL    RDYWT   ; AWAIT XFER COMPLETION
011C C30001                     JMP     XLOOP   ; BACK TO TRANSFER LOOP

011F 3E04               ENDIT:  MVI     A,04H   ; 4 IS END-OF-XFER SIGNAL
0121 D301                       OUT     1       ; SEND IT TO SIGNAL FINIS
0123 CD2901                     CALL    RDYWT
0126 C30500                     JMP     5       ; BACK TO MONITOR


0129 AF                 RDYWT:  XRA     A       ; CLEAR A TO
012A D303                       OUT     3       ; SELECT CONTROL PORT
012C DB01                       IN      1       ; READ IT
012E B7                         ORA     A       ; SET FLAGS
012F F22901                     JP      RDYWT   ; & TEST FOR DONE BIT HI

0132 C9                         RET             ; WHEN IT'S HI, BACK TO CALLER

0133 AF                 INTR11: XRA     A       ; GET ZERO IN ACCUM
0134 D303                       OUT     3       ;   TO SELECT CSR PORT
0136 3E04                       MVI     A,4     ; SET BIT 6 ON 11 CSR HI
0138 D301                       OUT     1       ; TO GENERATE INTERRUPT
013A C9                         RET             ; BACK TO CALLER

                        ; UPPER TO LOWER CASE CONVERSION ROUTINE

013B FE41               UTOL:   CPI     41H     ; 41 HEX = 'A'
013D F8                         RM              ; IF < 'A' RETURN
013E FE5B                       CPI     5BH     ; 5B HEX = 'Z'
0140 F0                         RP              ; IF > 'Z' RETURN

                        ; IF WE GET HERE IT'S AN ALPHA CHARACTER
```

```
                        ;    & NEEDS CNVERSION TO LOWER CASE

0141 F620                       ORI     20H     ; MASK IN BIT 6 HI
0143 C9                         RET             ; CONVERSION COMPLETE

                        ; CHANGE CARRIAGE RETURN TO LINE FEED & ECHO LF

0144 FE0D               CRLF:   CPI     0DH     ; 0D HEX = CARRIAGE RETURN
0146 C0                         RNZ             ; IF NOT CR, FORGET IT

0147 0E0A                       MVI     C,0AH   ; SEND LINEFEED TO CONSOLE
0149 CD5901                     CALL    CONOT   ; CONOT WILL PUT LF IN ACCUM
014C C9                         RET             ;   & IT WILL BE SENT TO UNIX

                        ; CONSOLE INPUT ROUTINE ; READ ONE CHAR FROM CONSOLE

014D DB70               CONIN:  IN      CSTAT
014F E601                       ANI     CKBR
0151 CA4D01                     JZ      CONIN

0154 DB71                       IN      CDATA
0156 E67F                       ANI     7FH
0158 C9                         RET

                        ; CONSOLE OUTPUT ROUTINE ; SEND ONE CHAR TO CONSOLE

0159 DB70               CONOT:  IN      CSTAT
015B E602                       ANI     CPTR
015D CA5901                     JZ      CONOT
0160 79                         MOV     A,C
0161 D371                       OUT     CDATA
0163 C9                         RET

0164                            END     100H
```

APPENDIX 4

UNIX DRIVER PREFIX FILE

```
#
/*      Prefix file for all MC driver code                      */

#define MC      0170424 /* Address of interface on Unibus */

struct                  /* Used to access Unibus registers */
{
        int     csr;    /* Control/status register      */
        int     buf;    /* Data register */
}       ;

#define RUN     1       /* Strobe bit in MC.csr */
#define DONE    0200    /* Interrupt enable bit in MC.csr */
#define ERROR   0100000 /* Error occurred during transfer */

struct                  /* Control block for interface driver routines */
{
        int     stat;   /* Status word */
        char    cmd;    /* Type of transfer , no longer used */
}       mctab;

#define ACTIVE  1       /* Transfer in progress flag */
#define READING 2       /* Reading from interface */
#define WRITING 4       /* Writing to interface */
#define LOGGED  010     /* Process is logged in */

#define MCRD    1       /* Command code for READ transfer */
#define MCWR    -1      /*     "       "   "  WRITE transfer */
#define MCPRI   30      /* Sleep priority during interrupt waits */

#define ever    (;;)    /* Used at top of infinite loop (for ever) */

#define MCDELAY 36      /* Delay (in Hertz) to simulate Baud rate interrupt */
```

APPENDIX 5

UNIX TEST DRIVER, SELF LOGIN

```
#include        "param.h"
#include        "user.h"
#include        "mc.h"

char    logstr[] "jtm\n" ;
char    funstr[] "test\n" ;

mci150()
{
        if((mctab.stat & ACTIVE) == 0)
                mctab.stat =| ACTIVE ;  /* Show it as active */

        wakeup(&mctab.stat);
        return;
}

mci154()
{
        if(mctab.stat & ACTIVE)
                wakeup(&mctab.stat) ;
}

mcread()
{
        static int      idp,fp ;

        if((mctab.stat & ACTIVE) == 0)
                sleep(&mctab.stat,MCPRI) ;
        else
        {
                if((&mctab.stat & LOGGED) == 0)
                        mclogin();      /* Fake the login */
                else
                        while(passc(getmc()) >= 0);
        }
}

mcwrite()
{
        if(mctab.stat&WRITING)  /* Only write if process logged in */

                while(putmc(mcpass())) > 0);    /* -1 returned at write end */
}

/*      This routine takes care of the user structure stuff for each
*       character passed during a write. Routine duplicates system
*       function cpass() but doesn't convert data to ASCII
*/

/*      Don't need this thing */
mcpass()

{
register  char c;

        if(u.u_count == 0)      /* u.u_count goes to 0 on last character */
                return(-1);     /* Signal upper level w/ negative return */
```

```
        c = *u.u_base;          /* Get a character from user's buffer */
        u.u_count--;            /* Decrement remaining character count */
        if(++u.u_offset[1] == 0) /* Update last char used position offset */
                u.u_offset[0]++;
        u.u_base++;             /* Point to next user character */
        return(c);              /* Pass character up for use */
}
/*      Don't need it at all */

/*
*       Routine to pass a character across interface. Usually this character
*       comes from user's own buffer via user structure "u" (in "user.h").
*/

putmc(c)

char    c;

{
        MC->buf = c;    /* Put the character in buffer register */
        MC->csr =| RUN|IENABL;  /* Allow other end to wake us up */

        sleep(&mctab.stat,MCPRI);  /* and then snooze till then */

/*   After interrupt & wakeup, calling process continues */

}

/*
*       Routine to pick up characters passed from micro & pass them up */
*/

getmc()
{

        MC->csr =| RUN|IENABL;
        sleep(&mctab.stat,MCPRI);
a
        return(MC->buf&0377);
}

mclogin()       /* Routine to send login info for process */
{
        static int fp,idp;
        if(logstr[idp] != '\0' )
                passc(logstr[idp++]);
        else
        {
                if(funstr[fp] != '\0' )
                {
                        if(funstr[fp] == '\n' )
                                mctab.stat =| LOGGED;
                        passc(funstr[fp++]);
                }
        }
}
```

APPENDIX 6

UNIX TEST DRIVER

```
#include        "param.h"
#include        "user.h"
#include        "mc.h"

char    logstr[] "jtm\n" ;
char    funstr[] "test\n" ;

extern lbolt;   /* System function that wakes up every 2 secs */

mci150()
{
        if((mctab.stat & ACTIVE) == 0)
                mctab.stat =| ACTIVE ;  /* Show it as active */

        wakeup(&mctab.stat);
        return;
}

mci154()
{
        return; /* This vector can't be used; toss interrupt */
}

mcopen()
{
        mctab.stat =| WRITING;
}

mcclose()
{
        mctab.stat =& ~WRITING;
}

mcread(()
{

register char c;

        if((mctab.stat & ACTIVE) == 0)  /* If not logged in */
                sleep(&mctab.stat,MCPRI) ;  /* Wait for interrupt */
        else
        {
                do                      /* Stay in this loop till count = 0 */
                {
                        c = getmc();    /* Pick up a character */

                        if(c == '\004') /* End of login requested */
                                mctab.stat =& ~ACTIVE;

                        if(c == '\r') /* Convert CR to LF */
                                c = '/n'; /*  for Shell's benefit */

                }while(passc(c) >= 0);  /* -1 returned when count = 0 */

mcwrite()
{
        if(mctab.stat&WRITING)  /* Only write if process logged in */
```

```
                while((putmc9cpass())) > 0); -1 returned at write end */
}


/*
*       Routine to pass a character across interface. Usually this character
*       comes from user's own buffer via user structure "u" (in "user.h").
*/

putmc(c)

char    c;

{
        MC->buf = c;    /* Put the character in buffer register */
        MC->csr =| RUN; /* Allow other end to wake us up */

        while(!(MC->csr & DONE))        /* Wait for xfer completion */
                sleep(&lbolt,MCPRI);    /* Look in every 2 secs */

/*   After interrupt & wakeup, calling process continues */

}

/*
*       Routine to pick up characters passed from micro & pass them up */
*/

getmc()
{
        MC->csr =| RUN;                 /* Turn on interface */
        while(!(MC->csr & DONE))        /* Wait for xfer completion signal */
                sleep(&lbolt,MCPRI);    /* Look in every 2 secs for signal */

        return(MC->buf&0377);           /* Send character up to caller */
}
```

APPENDIX 7

UNIX DRIVER, FINAL VERSION

```
#include        "param.h"
#include        "user.h"
#include        "mc.h"

mci150()
{
        return;
}

mci154()
{
        return; /* This vector can't be used; toss interrupt */
}

mcopen()
{
        mctab.stat =| WRITING;  /* That's all we need to do */
}

mcclose()
{
        mctab.stat =& ~WRITING; /* Don't let output go to interface now */
}

mcread(()
{

register char c;

        while(passc(getmc()) >= 0;      /* -1 returned when count = 0 */
}

mcwrite()
{
        if(mctab.stat&WRITING)  /* Only write if process logged in */

                while((putmc(cpass())) > 0); -1 returned at write end */
}

/*   This thing just wakes up the sleeping getmc() & putmc() tasks */

timesup()
{
        wakeup(&mctab.stat);
}

/*
*       Routine to pass a character across interface. Usually this character
*       comes from user's own buffer via user structure "u" (in "user.h").
*/

putmc(c)

char    c;

{
        MC->buf = c;    /* Put the character in buffer register */
```

```
        MC->csr =| RUN; /* Allow other end to wake us up */

        while(!(MC->csr & DONE))        /* Wait for xfer completion */
        {
                timeout(timesup,NULL,MCDELAY); /* Schedule wakeup */
                sleep(&lbolt,MCPRI);    /* Look in occasionally */
        }

/*   After interrupt & wakeup, calling process continues */

}

/*
*       Routine to pick up characters passed from micro & pass them up */
*/

getmc()
{
        MC->csr =| RUN;         /* Turn on interface */

        while(!(MC->csr & DONE))        /* Wait for xfer completion */
        {
                timeout(timesup,NULL,MCDELAY); /* Schedule wakeup */
                sleep(&lbolt,MCPRI);    /* Look in occasionally */
        }

        return(MC->buf80377);           /* Send character up to caller */
}
```

APPENDIX 8

UNIX USER TEST PROGRAM

```
#include        "param.h"
#include        "user.h"

extern errno;
char    errstr[] "Can't open mc -> ";
int     errstcnt;

main()
{
        char *bp,*err;
        int fp,mcp;

        errstcnt = 19;

/* Open data file to store characters passed. If not there create it */

        fp = open("data",1);
        if(fp == -1)
                fp = creat("data",0644);

/* Open interface */

        mcp = open("/dev/ttym",2);

/* If interface can't be opened, send error number to file */

        if(mcp == -1)
        {
                *err = errno + '0';
                write(fp,errstr,errstcnt);
                write(fp,err,1);
                exit();
        }

        for(;;)         /* Exit is inside this forever loop */
        {
                read(mcp,bp,1);

                if(*bp == 0)
                        exit();

                write(fp,bp,1);
        }
}
```

APPENDIX 9

CP/M DISK CREATE PROGRAM

```
                        *****   CREATE A VIRTUAL DISK ON UNIX  *****
                        *
                        *       THIS PROGRAM PICKS UP A USER DEFINED
                        *       VIRTUAL DISK NAME OF UP TO 10 CHARACTERS
                        *       AND PASSES THE NAME TO UNIX.  THE NAME
                        *       IS USED TO CREATE THE INDEX AND DATA
                        *       FILES TO BE ASSOCIATED WITH THAT
                        *       VIRTUAL DISK.  MOUNT11 MUST BE RUN
                        *       EACH TIME THE DISK IS TO BE ACCESSED.
                        *
0100                            ORG     100H            TRANSIENT ROUTINE
                        *
0005 =                          CPM     EQU     5       ;MONITOR RETURN ADDRESS
7AFF =                          CONIN   EQU     7AFFH   ;CP/M CONSOLE IN ROUTINE
7B15 =                          CONOT   EQU     7B15H   ;CP/M CONSOLE OUT ROUTINE
7D20 =                          PMSG    EQU     7D20H   ;CP/M MESSAGE PRINT ROUTINE
7E2E =                          CRD11   EQU     7E2EH   ;CP/M PDP11 READ ROUTINE
7E50 =                          CWR11   EQU     7E50H   ;CP/M PDP11 WRITE ROUTINE
                        *
                        *
0100 215101                     LXI   H,ENTMSG          ;GET ADDRESS OF ENTER MSG IN H
0103 CD207D                     CALL  PMSG              ;PRINT 'ENTER' MESSAGE

0106 060B                       MVI   B,0BH             ;SET COUNT TO 10 CHARS+CR
0108 21A301                     LXI   H,NAMBUF          ;HL->NAME BUFFER HEAD

010B CDFF7A             INLOOP: CALL  CONIN             ;PICK UP A CHARACTER
010E FE03                       CPI   3                 ;IS IT CNTL-C ?
0110 CA0500                     JZ    CPM               ;IF SO,BACK TO MONITOR

0113 77                         MOV   M,A               ;OTHERWISE, SAVE IT
0114 4F                         MOV   C,A               ;FOR CONOT
0115 CD157B                     CALL  CONOT             ;ECHO THE CHARACTER TO CONSOLE

0118 79                         MOV   A,C               ;GET CHAR BACK IN ACCUM
0119 FE0D                       CPI   0DH               ;WAS IT CARRIAGE RETURN ?
011B CA2C01                     JZ    SENDIT            ;IF SO, WE'RE DONE GETTING NAME

011E 23                         INX   H                 ;POINT TO NEXT MEMORY LOCATION
011F 05                         DCR   B                 ;COUNT > 10 ?
0120 F20B01                     JP    INLOOP            ;IF NOT, GET MORE CHARS
                        *
                        *   ONLY GET HERE IF MORE THAN 10 CHARS ENTERED FOR NAME
                        *
0123 217B01                     LXI   H,ERRMSG          ;HL->ERROR MSG BUFFER HEAD
0126 CD207D                     CALL  PMSG              ;PRINT ERROR MESSAGE
0129 C30500                     JMP   CPM               ;BACK TO MONITOR
                        *
                        *   SEND NAME TO UNIX
                        *
012C 0E0A               SENDIT: MVI   C,0AH             ;SEND A LINE-FEED TO CONOT
012E CD157B                     CALL  CONOT             ; TO BE ECHOED TO CONSOLE

0131 3600                       MVI   M,0H              ;MARK END OF NAME WITH ZERO
0133 219901                     LXI   H,UNICMD          ;HL -> UNIX FILE NAME
```

```
                CMDLOP:
0136 CD507E             CALL CWR11              ;SEND A CHARACTER TO UNIX


0139 23                 INX  H                  ;POINT TO NEXT CHARACTER
013A AF                 XRA  A                  ;GET ZERO FOR COMPARE
013B BE                 CMP  M                  ;END OF NAME ?
013C C23601             JNZ  CMDLOP             ;IF NOT CONTINUE

013F 21A301             LXI  H,NAMBUF           ;HL->NAME BUFFER HEAD

                LOOP11:
0142 CD507E             CALL CWR11              ;SEND THE CHARACTER TO UNIX
0145 AF                 XRA  A                  ;GET ZERO IN ACCUM FOR COMPARE
0146 BE                 CMP  M                  ;IS BUFFER CONTENTS = 0 ?
0147 CA4E01             JZ   ENDIT              ;IF SO GET RETURN CODE

014A 23                 INX  H                  ;IF NOT POINT TO NEXT CHAR
014B C34201             JMP  LOOP11             ;  AND SEND IT


                ENDIT:
014E C30500             JMP  CPM                ;NOTHING MORE TO DO
                *                               ;  GO BACK TO MONITOR
0151 454E544552ENTMSG:  DB   'ENTER VIRTUAL DISK NAME (10 CHARS MAX) > ',0H
017B 43414E4E4FERRMSG:  DB   'CANNOT MOUNT REQUESTED DISK',0DH,0AH,0H
0199 4D414B4544UNICMD:  DB     'MAKEDISK',0DH,0H
01A3 0B         NAMBUF: DB   11


01A4                    END  100H
```

APPENDIX 10

UNIX VIRTUAL DISK CREATE PROGRAM

```
/*
 *    Routine to create and initialize virtual floppy disk files
 *    for use with UNIX/CPM interface.  Files needed for a single
 *    virtual floppy are
 *       1) index file - name is 'xxxxxxxxx.ind' , where xxxxxxxxx
 *                       is CP/M user supplied disk name.  Contents
 *                       of file are track-sector pairs used as
 *                       unique identifiers for sectors on the
 *                       virtual disk.  The location of a given pair
 *                       in the index file is used to find the offset
 *                       to the desired 128 byte sector in the data
 *                       file for that virtual disk.
 *
 *       2) data file -  name is 'xxxxxxxxx.dat'. Contents of
 *                       file are 128 byte blocks of data corresponding
 *                       one-to-one with a unique sector on the virtual
 *                       disk.  Data blocks are contiguous and can only
 *                       be separated by calculating the offset to a
 *                       given block.
 *
 *
 *    This program must be run once for each virtual disk desired.
 *    Program is run by running CP/M program CREAT11 on the CP/M
 *    side of the interface. Subsequent requests to create an
 *    already existing virtual disk will result in re-initialization
 *    of the disk and loss of any data.  Once a disk is created,
 *    CP/M program MOUNT11 must be run to establish the UNIX/CPM
 *    link needed for virtual I/O.  After MOUNT11 has been run,
 *    all CP/M I/O referencing a non-existent disk drive on the
 *    CP/M system will result in I/O to the 'mounted' virtual disk
 *    under UNIX.  No further actions are necessary to do the
 *    virtual I/O.  When the CP/M user is done using the virtual
 *    disk, he/she must run UMOUNT11 under CP/M to break the link
 *    between the two systems and clear the way for another MOUNT11
 *    request.
 */

main()
{
   char *name,*tname,*index,*data;
   char direntry[2];              /* To hold directory track/sector pair */
   int  length,mcp,ip,dp,i;

   length = 0;                    /* Initialize user disk name length */
   direntry[0] = 2;               /* Directory starts on track 2, */
   direntry[1] = 1;               /*    sector 1 */

   mcp = open("/dev/ttym",2);     /* Open the interface to CP/M */

   name = tname;                  /* Save pointer to name buffer head */

   do   {
        read(mcp,tname,1);        /* Pick up characters of name one at a time */
        length++;                 /*   and keep count of them */
   }while(*tname++);              /* Quit when a null character is passed */

   length--;                      /* Skip null character */
```

```
    concat(index,name,".ind",length,5); /* Make index file name */
    ip = creat(index,0666);       /* Create (or truncate) the index file */

    concat(data,name,".dat",length,5); /* Make data file name */
    dp = creat(data,0666);       /* Create (or truncate) data file */

    write(ip,direntry,2);         /* Set up directory track/sector entry */
    for(i=1;i<=128;i++)
         write(dp,'\0',1);        /* Zero out directory for CP/M's benefit */
}

/*
 *   Routine to concatenate two character strings into a third location
 *   Arguments are:
        s1       name of new (concatenated) string
        s2       name of first string (left side of concatenation)
        s3       name of second string (right side of concatenation)

        c1       length of first string
        c2       length of second string
 */

concat(s1,s2,s3,c1,c2)

    char *s1,*s2,*s3;
    int  c1,c2;
{
    while(c1--)
         *s1++ = *s2++    /* Move first string into new string  */

    while(c2--)
         *s1++ = *s3++    /* Concatenate second string onto new string */
}
```

APPENDIX 11

UNIX DISK CREATE DEBUGGER PROGRAM

```
/*
 *    Routine to create and initialize virtual floppy disk files
 *    for use with UNIX/CPM interface.  Files needed for a single
 *    virtual floppy are
 *       1) index file - name is 'xxxxxxxxxx.ind' , where xxxxxxxxxx
 *                       is CP/M user supplied disk name.  Contents
 *                       of file are track-sector pairs used as
 *                       unique identifiers for sectors on the
 *                       virtual disk.  The location of a given pair
 *                       in the index file is used to find the offset
 *                       to the desired 128 byte sector in the data
 *                       file for that virtual disk.
 *
 *       2) data file -  name is 'xxxxxxxxxx.dat'. Contents of
 *                       file are 128 byte blocks of data corresponding
 *                       one-to-one with a unique sector on the virtual
 *                       disk.  Data blocks are contiguous and can only
 *                       be separated by calculating the offset to a
 *                       given block.
 *
 *
 *    This program must be run once for each virtual disk desired.
 *    Program is run by running CP/M program CREAT11 on the CP/M
 *    side of the interface. Subsequent requests to create an
 *    already existing virtual disk will result in re-initialization
 *    of the disk and loss of any data.  Once a disk is created,
 *    CP/M program MOUNT11 must be run to establish the UNIX/CPM
 *    link needed for virtual I/O.  After MOUNT11 has been run,
 *    all CP/M I/O referencing a non-existent disk drive on the
 *    CP/M system will result in I/O to the 'mounted' virtual disk
 *    under UNIX.  No further actions are necessary to do the
 *    virtual I/O.  When the CP/M user is done using the virtual
 *    disk, he/she must run UMOUNT11 under CP/M to break the link
 *    between the two systems and clear the way for another MOUNT11
 *    request.
 */

main()
{
   char *name,*tname,*index,*data;
   char direntry[2];              /* To hold directory track/sector pair */
   int  length,mcp,ip,dp,i;

   length = 0;                    /* Initialize user disk name length */
   direntry[0] = 2;               /* Directory starts on track 2, */
   direntry[1] = 1;               /*    sector 1 */

   mcp = open("/dev/ttyb",2);     /* Open the interface to CP/M */

   name = tname;                  /* Save pointer to name buffer head */

   do   {
        read(mcp,tname,1);        /* Pick up characters of name one at a time */
        length++;                 /*   and keep count of them */
   }while(*tname++);              /* Quit when a null character is passed */

   length--;                      /* Skip null character */
```

```
    concat(index,name,".ind",length,5); /* Make index file name */
    ip = creat(index,0666);       /* Create (or truncate) the index file */

    concat(data,name,".dat",length,5); /* Make data file name */
    dp = creat(data,0666);        /* Create (or truncate) data file */

    write(ip,direntry,2);         /* Set up directory track/sector entry */
    for(i=1;i<=128;i++)
         write(dp,'\0',1);        /* Zero out directory for CP/M's benefit */
}

/*
 *   Routine to concatenate two character strings into a third location
 *   Arguments are:
        s1      name of new (concatenated) string
        s2      name of first string (left side of concatenation)
        s3      name of second string (right side of concatenation)

        c1      length of first string
        c2      length of second string
 */

concat(s1,s2,s3,c1,c2)

    char *s1,*s2,*s3;
    int  c1,c2;
{
    while(c1--)
         *s1++ = *s2++   /* Move first string into new string  */

    while(c2--)
         *s1++ = *s3++   /* Concatenate second string onto new string */
}
```

APPENDIX 12

CP/M DISK MOUNT REQUEST PROGRAM

```
                ;*****   MOUNT A VIRTUAL DISK ON UNIX  *****
                ;
                ;       THIS PROGRAM PICKS UP A USER DEFINED
                ;       VIRTUAL DISK NAME OF UP TO 10 CHARACTERS
                ;       AND PASSES THE NAME TO UNIX.  THE NAME
                ;       IS USED TO REFERENCE THE INDEX AND DATA
                ;       FILES FOR THAT DISK, PROVIDED THE FILES
                ;       ALREADY EXIST.  IF THEY DO NOT EXIST,
                ;       UNIX RETURNS AN ERROR MESSAGE AND THE
                ;       DISK MUST BE CREATED USING 'CREAT11'.
                ;
                ;       ONCE A DISK IS MOUNTED, ALL CP/M I/O
                ;       DIRECTED TO DISK C OR D WILL BE DONE ON
                ;       THE VIRTUAL DISK VIA UNIX.
                ;
0100                    ORG     100H            TRANSIENT ROUTINE
                ;
0005 =          CPM     EQU     5       ;MONITOR RETURN ADDRESS
7AFF =          CONIN   EQU     7AFFH   ;CP/M CONSOLE IN ROUTINE
7B15 =          CONOT   EQU     7B15H   ;CP/M CONSOLE OUT ROUTINE
7D20 =          PMSG    EQU     7D20H   ;CP/M MESSAGE PRINT ROUTINE
7E2E =          CRD11   EQU     7E2EH   ;CP/M PDP11 READ ROUTINE
7E50 =          CWR11   EQU     7E50H   ;CP/M PDP11 WRITE ROUTINE
                ;
                ;
0100 215101             LXI   H,ENTMSG          ;GET ADDRESS OF ENTER MSG IN H
0103 CD207D             CALL  PMSG              ;PRINT 'ENTER' MESSAGE

0106 060B               MVI   B,0BH             ;SET COUNT TO 10 CHARS+CR
0108 21A301             LXI   H,NAMBUF          ;HL->NAME BUFFER HEAD

010B CDFF7A     INLOOP: CALL  CONIN             ;PICK UP A CHARACTER
010E FE03               CPI   3                 ;IS IT CNTL-C ?
0110 CA0500             JZ    CPM               ;IF SO,BACK TO MONITOR

0113 77                 MOV   M,A               ;OTHERWISE, SAVE IT
0114 4F                 MOV   C,A               ;FOR CONOT
0115 CD157B             CALL  CONOT             ;ECHO THE CHARACTER TO CONSOLE

0118 79                 MOV   A,C               ;GET CHAR BACK IN ACCUM
0119 FE0D               CPI   0DH               ;WAS IT CARRIAGE RETURN ?
011B CA2C01             JZ    SENDIT            ;IF SO, WE'RE DONE GETTING NAME

011E 23                 INX   H                 ;POINT TO NEXT MEMORY LOCATION
011F 05                 DCR   B                 ;COUNT > 10 ?
0120 F20B01             JP    INLOOP            ;IF NOT, GET MORE CHARS
                ;
                ;   ONLY GET HERE IF MORE THAN 10 CHARS ENTERED FOR NAME
                ;
0123 218901             LXI   H,ERRMSG          ;HL->ERROR MSG BUFFER HEAD
0126 CD207D             CALL  PMSG              ;PRINT ERROR MESSAGE
0129 C30500             JMP   CPM               ;BACK TO MONITOR
                ;
                ;   SEND NAME TO UNIX
                ;
```

```
012C 0E0A       SENDIT: MVI   C,0AH             ;SEND A LINE-FEED TO CONOT
012E CD157B             CALL  CONOT             ; TO BE ECHOED TO CONSOLE

0131 3600               MVI   M,0H              ;MARK END OF NAME WITH ZERO
0133 21A701             LXI   H,UNICMD          ;HL -> UNIX FILE NAME
               CMDLOP:
0136 CD507E             CALL  CWR11             ;SEND A CHARACTER TO UNIX


0139 23                 INX   H                 ;POINT TO NEXT CHARACTER
013A AF                 XRA   A                 ;GET ZERO FOR COMPARE
013B BE                 CMP   M                 ;END OF NAME ?
013C C23601             JNZ   CMDLOP            ;IF NOT CONTINUE

013F 21B101             LXI   H,NAMBUF          ;HL->NAME BUFFER HEAD

               LOOP11:
0142 CD507E             CALL  CWR11             ;SEND THE CHARACTER TO UNIX
0145 AF                 XRA   A                 ;GET ZERO IN ACCUM FOR COMPARE
0146 BE                 CMP   M                 ;IS BUFFER CONTENTS = 0 ?
0147 CA4E01             JZ    ENDIT             ;IF SO GET RETURN CODE

014A 23                 INX   H                 ;IF NOT POINT TO NEXT CHAR
014B C34201             JMP   LOOP11            ;  AND SEND IT


014E CD2E7E     ENDIT:  CALL  CRD11             ;UNIX WILL SEND RETURN CODE
0151 AF                 XRA   A                 ;ZERO ACCUM FOR COMPARE
0152 BE                 CMP   M                 ;WHAT WAS RETURN CODE ?
0153 CA0500             JZ    CPM               ;IF ZERO RETURN WE'RE OK

0156 218901             LXI   H,ERRMSG          ;IF NOT ZERO,PRINT BAD REQ MSG
0159 CD207D             CALL  PMSG              ;GO TO PRINT ROUTINE

015C C30500             JMP   CPM               ;  AND GO BACK TO MONITOR
               *
015F 454E544552ENTMSG:  DB    'ENTER VIRTUAL DISK NAME (10 CHARS MAX) > ',0H
0189 43414E4E4FERRMSG:  DB    'CANNOT MOUNT REQUESTED DISK',0DH,0AH,0H
01A7 5649525446UNICMD:  DB      'VIRTFLOP',0DH,0H
01B1 0B         NAMBUF: DB    11


01B2                    END   100H
```

APPENDIX 13

UNIX VIRTUAL DISK PROGRAM

```
#define READ    1       /* read request command from CP/M */
#define WRITE   -1      /* write request command from CP/M */
#define EXIT    0       /* exit request from CP/M */
#define ever    (;;)    /* Infinite loop condition for 'for ever' */


/*
 *   Virtual Floppy Disk Simulation Program
 *   This program simulates a floppy disk and drive for CP/M
 *   format disks.
 */

main()
{
   char *data,*index,*bp,track,sector,*ind,*name,*tname;

   int  mcp,indexfp,datafp,length,count,offset,command,supoffset;


   length = 0;          /* Initialize file name length indicator */

   mcp = open("/dev/ttym",2);   /* Open interface to CP/M */

   name = tname;                /* Save ptr to head of file name buffer */
   do   {
        read(mcp,tname,1);      /* Pick up characters of name one at a time */
        length++;               /*   and keep count of them */
   }while(*tname++);            /* Quit when a null character is passed */

   length--;                    /* Skip null character */

   concat(index,name,".ind",length,5); /* Make index file name (name.ind) */

   indexfp = open(index,2);     /* Open track/sector index file */
   if(indexfp == -1)            /* If file can't be opened or doesn't exist */
        mcerr(mcp);             /*    quit with error */

   concat(data,name,".dat",length,5); /* Make data file name (name.dat) */

   datafp = open(data,2);       /* Open virtual data file */
   if(datafp == -1)             /* If it can't be opened or doesn't exist */
        mcerr(mcp);             /*    quit with error */
   else
        write(mcp,'\0',1);      /* If all OK signal CP/M to go ahead */

/*
 *   The following code is executed indefinitely until the CP/M user
 *   runs the UMOUNT program. UMOUNT sends a null character as the
 *   next command, which causes this program to terminate and close
 *   all files (automatic in Unix).  The user must then run CP/M program
 *   MOUNT11 to start this program up again from the top, opening all
 *   files again and performing virtual "I/O" until UMOUNT11 is run
 *   again from CP/M
 */

   for ever     {               /* Stay in this loop until exit requested */
```

```
        read(mcp,&command,1);    /* Pick up CP/M request code */
                                 /* Codes are 1:READ, -1:WRITE, 0:EXIT */
        if(command == EXIT)
                exit();          /* Close the interface and exit */

        if(command != READ && command != WRITE) /* Check for bad command */
                mcerr(mcp);      /* Exit with error if command undefined */

/*    Pick up track and sector from CP/M. These will be used as an index
 *    into the track/sector index file.
 */

        read(mcp,&track,1);      /* Get requested track */
        read(mcp,&sector,1);     /*     and sector numbers */
/*
 *    Here we search through the index file for the track/sector
 *    pair in the request. Simultaneously, we keep track of the offset
 *    into the index file with integer 'offset'.  This integer will be
 *    used to seek into the data file for the requested sector.  Since
 *    each sector is exactly 128 bytes long, the offset to the pair in
 *    the index file will be multiplied by 128 to get the offset to the
 *    start of the requested sector in the data file.
 */

        offset = 0;              /* Reinitialize offset counter */

        while(count = read(indexfp,ind,2); /* Read till end-of-file */
        {
           if(*ind == track)             /* If tracks match */
           {
                if(*(ind+1) == sector)   /*  and sectors match too */
                        goto found;      /*   jump out of read loop */
           }
           offset++;                     /* Otherwise just increment offset */
        }

/*
 *    We get here if no match was found.  This is OK in the case of a
 *    WRITE request, but blatantly wrong in the case of a READ.
 *    If it is a WRITE request, we create a new entry for the new
 *    track/sector pair.
 */
        if(command == WRITE)
        {
                seek(indexfp,0,2);       /* Move to end of file */
                write(indexfp,&track,1); /* Put the index track in file */
                write(indexfp,&sector,1); /*  and index sector */
        }
        else
                mcerr(mcp);              /* Don't tolerate undefined reads */
/*
 *    Here we go about the business of looking for the 128 byte sector in
 *    the data file (name.dat).  Since the file can be larger than 65535
 *    characters, we don't use a single integer for a single seek into
 *    the file.  Instead, we make use of the offset*512 option in the Unix
 *    seek command.  The offset calculated during the index file search
 *    is broken into two parts.  The first (supoffset) is = (offset*128)/512
 *    This offset is used to do the required number ot 512 byte seeks.
 *    Then the original offset is reset to its modulus 4 (512/128) value
 *    and multiplied by 128.  This new offset is used for the remainder of
 *    the seek.
```

```
 */

found:  if(offset >= 4)          /* Offset is 'real offset'/128; 512/128=4 */
        {
                supoffset = offset >>2; /* Only want offset portion >= 512 */
                seek(datafp,supoffset,3); /* The 3 means seek in 512 byte */
                                          /* increments, supoffset times */
                offset =% 4;     /* Adjust offset to < 512 after multiply */
        }

        offset =* 128;           /* Expand offset for single byte seek */
        seek(datafp,offset,1);   /* The 1 means seek from current position */
                                 /*    to current position + offset bytes */

/*
 *    Now we do the actual I/O operation for CP/M.  Datafp is pointed to
 *    the start of the requested 128 byte sector; reading and writing will
 *    proceed from that location in the data file.
 *    Note that command was verified at top of forever loop, so if-else
 *    test will do here.
 */
        if(command == READ)      /* Code for executing CP/M read request */
        {
                read(datafp,bp,128); /* Get the sector from data file */
                write(mcp,bp,128);   /* Write the sector to CP/M */
        }
        else                     /* Code for executing CP/M write request */
        {
                read(mcp,bp,128);  /* Get the 128 bytes from CP/M */
                write(datafp,bp,128);    /* Write the sector into data file */
        }

        seek(datafp,0,0);    /* Reset pointer to start of file for more I/O */
        seek(indexfp,0,0);   /* Rewind index file also */

    }     /*    End of forever loop  */

}         /*    End of main  */

mcerr(ip)

int   ip;
{       write(ip,'\0377',1);     /* Send error indicator to CP/M */
        close(ip);               /* Close the interface */
        exit();                  /*    and quit */
}

/*
 *    Routine to concatenate two character strings into a third location
 *    Arguments are:
        s1      name of new (concatenated) string
        s2      name of first string (left side of concatenation)
        s3      name of second string (right side of concatenation)

        c1      length of first string
        c2      length of second string
 */

concat(s1,s2,s3,c1,c2)

   char *s1,*s2,*s3;
```

```
    int  c1,c2;
{
    while(c1--)
         *s1++ = *s2++    /* Move first string into new string  */

    while(c2--)
         *s1++ = *s3++    /* Concatenate second string onto new string */
}
```

APPENDIX 14

UNIX VIRTUAL DISK DEBUGGER PROGRAM

```
#define READ     '1'      /* read request command from CP/M */
#define WRITE    '2'      /* write request command from CP/M */
#define EXIT     '0'      /* exit request from CP/M */
#define ever     (;;)     /* Infinite loop condition for 'for ever' */
#define WRITING 4         /* Write goes to interface when set */


/*
 *    Virtual Floppy Disk Simulation Program
 *    This program simulates a floppy disk and drive for CP/M
 *    format disks.
 */

main()
{
   char *data,*index,track,sector,*ind,*name,*tname;
   char iobuff[128];

   int  mcp,indexfp,datafp,length,count,offset,command,supoffset;


   length = 0;             /* Initialize file name length indicator */

   mcp = open("/dev/ttyb",2);    /* Open interface to CP/M */

   name = tname;                 /* Save ptr to head of file name buffer */
   do   {
        read(mcp,tname,1);       /* Pick up characters of name one at a time */
        length++;                /*   and keep count of them */
   }while(*tname++ != '0')       /* Quit when a null character is passed */

   length--;                     /* Skip null character */

   concat(index,name,".ind",length,5); /* Make index file name (name.ind) */

   indexfp = open(index,2);      /* Open track/sector index file */
   if(indexfp == -1)             /* If file can't be opened or doesn't exist */
        mcerr(mcp);              /*    quit with error */

   concat(data,name,".dat",length,5); /* Make data file name (name.dat) */

   datafp = open(data,2);        /* Open virtual data file */
   if(datafp == -1)              /* If it can't be opened or doesn't exist */
        mcerr(mcp);              /*    quit with error */
   else
        write(mcp,"Good request\n",13); /* If all OK signal CP/M to go ahead */

/*
 *   The following code is executed indefinitely until the CP/M user
 *   runs the UMOUNT program. UMOUNT sends a null character as the
 *   next command, which causes this program to terminate and close
 *   all files (automatic in Unix).  The user must then run CP/M program
 *   MOUNT11 to start this program up again from the top, opening all
 *   files again and performing virtual "I/O" until UMOUNT11 is run
 *   again from CP/M
 */
```

```
   for ever     {                   /* Stay in this loop until exit requested */

        read(mcp,&command,1);       /* Pick up CP/M request code */
                                    /* Codes are 1:READ, -1:WRITE, 0:EXIT */
        if(command == EXIT)
                exit();             /* Close the interface and exit */

        if(command != READ && command != WRITE) /* Check for bad command */
                mcerr(mcp);         /* Exit with error if command undefined */

/*    Pick up track and sector from CP/M. These will be used as an index
 *    into the track/sector index file.
 */

        read(mcp,&track,1);         /* Get requested track */
        read(mcp,&sector,1);        /*     and sector numbers */
/*
 *    Here we search through the index file for the track/sector
 *    pair in the request. Simultaneously, we keep track of the offset
 *    into the index file with integer 'offset'.  This integer will be
 *    used to seek into the data file for the requested sector.  Since
 *    each sector is exactly 128 bytes long, the offset to the pair in
 *    the index file will be multiplied by 128 to get the offset to the
 *    start of the requested sector in the data file.
 */

        offset = 0;                 /* Reinitialize offset counter */

        while(count = read(indexfp,ind,2); /* Read till end-of-file */
        {
           if(*ind == track)                /* If tracks match */
           {
                if(*(ind+1) == sector)      /*  and sectors match too */
                {
                   write(mcp,"\n",1);       /* Debug statements */
                   write(mcp,ind,1);
                   write(mcp,&track,1);
                   write(mcp,"\n",1);
                   write(mcp,ind+1,1);
                   write(mcp,&sector,1);
                   write(mcp,"\n",1);
                   goto found;      /*     jump out of read loop */
                }
           }
           offset++;                        /* Otherwise just increment offset */
        }

/*
 *    We get here if no match was found.  This is OK in the case of a
 *    WRITE request, but blatantly wrong in the case of a READ.
 *    If it is a WRITE request, we create a new entry for the new
 *    track/sector pair.
 */
        if(command == WRITE)
        {
                seek(indexfp,0,2);          /* Move to end of file */
                write(indexfp,&track,1); /* Put the index track in file */
                write(indexfp,&sector,1); /*  and index sector */
        }
        else
                mcerr(mcp);                 /* Don't tolerate undefined reads */
```

```
/*
 *   Here we go about the business of looking for the 128 byte sector in
 *   the data file (name.dat).  Since the file can be larger than 65535
 *   characters, we don't use a single integer for a single seek into
 *   the file.  Instead, we make use of the offset*512 option in the Unix
 *   seek command.  The offset calculated during the index file search
 *   is broken into two parts.  The first (supoffset) is = (offset*128)/512
 *   This offset is used to do the required number ot 512 byte seeks.
 *   Then the original offset is reset to its modulus 4 (512/128) value
 *   and multiplied by 128.  This new offset is used for the remainder of
 *   the seek.
 */

found:  if(offset >= 4)          /* Offset is 'real offset'/128; 512/128=4 */
        {
                supoffset = offset >>2; /* Only want offset portion >= 512 */
                seek(datafp,supoffset,3); /* The 3 means seek in 512 byte */
                                          /* increments, supoffset times */
                offset =& 03;    /* Adjust offset to < 512 after multiply */
        }

        offset =* 128;           /* Expand offset for single byte seek */
        seek(datafp,offset,1);   /* The 1 means seek from current position */
                                 /*    to current position + offset bytes */

/*
 *   Now we do the actual I/O operation for CP/M.  Datafp is pointed to
 *   the start of the requested 128 byte sector; reading and writing will
 *   proceed from that location in the data file.
 *   Note that command was verified at top of forever loop, so if-else
 *   test will do here.
 */
        if(command == READ)      /* Code for executing CP/M read request */
        {
                read(datafp,iobuff,128); /* Get the sector from data file */
                write(mcp,iobuff,128);   /* Write the sector to CP/M */
        }
        else                     /* Code for executing CP/M write request */
        {
                read(mcp,iobuff+count,128);  /* Get the 128 bytes from CP/M */
                write(datafp,iobuff,128);    /* Write the sector into data file */
        }

        seek(datafp,0,0);    /* Reset pointer to start of file for more I/O */
    \   seek(indexfp,0,0);   /* Rewind index file also */

   }    /*   End of forever loop  */

}       /*   End of main  */

mcerr(ip)

int   ip;
{       write(ip,"Can't do it\n",12);    /* Send error indicator */
        close(ip);               /* Close the interface */
        exit();                  /*   and quit */
}

/*
 *   Routine to concatenate two character strings into a third location
 *   Arguments are:
```

```
        s1      name of new (concatenated) string
        s2      name of first string (left side of concatenation)
        s3      name of second string (right side of concatenation)

        c1      length of first string
        c2      length of second string
 */

concat(s1,s2,s3,c1,c2)

   char *s1,*s2,*s3;
   int  c1,c2;
{
   while(c1--)
        *s1++ = *s2++   /* Move first string into new string  */

   while(c2--)
        *s1++ = *s3++   /* Concatenate second string onto new string */
}
```

APPENDIX 15

BIOS MODIFICATIONS, DISK SELECT CODE

```
                        ;
                        ; READ THE SECTOR AT SECT, FROM THE PRESENT TRACK
                        ; USE STARTING ADDRESS AT DMAADD.
                        ;
                        ; IF DISK C (OR D) IN USE FOR TO PDP-11 INTERFACE READ
                        ;
                        RDSEL:
                                IF    DISK11      ;IF VIRTUAL FLOPPY SYSTEM        <JTM>

7C0F 3A837E                     LDA   DISKNO      ;GET CURRENT DISK NUMBER         <JTM>
7C12 FE02                       CPI   2           ;IS IT DISK C ?                  <JTM>
7C14 F2177E                     JP    READ11      ;IF SO DO PDP-11 INTERFACE READ  <JTM>

                                ENDIF                                              <JTM>

                        ;
                        ;  READ FOR HARDWARE FLOPPY DISK
                        ;
                        ;
                        ; WRITE THE SECTOR AT SECT, ON THE PRESENT TRACK.
                        ; USE STARTING ADDRESS AT DMAADD.
                        ;
                        ;
                        WRSEL:
                                IF    DISK11      ;IF VIRTUAL FLOPPY SYSTEM        <JTM>

                        ; IF DISK C OR D IN USE GO TO PDP-11 INTERFACE WRITE      <JTM>
                        ;
7CC3 3A837E                     LDA   DISKNO      ;GET CURRENT DISK NUMBER         <JTM>
7CC6 FE02                       CPI   2           ;IS IT DISK C OR D ?             <JTM>
7CC8 F2397E                     JP    WRIT11      ;IF SO WRITE IS TO PDP-11        <JTM>

                                ENDIF                                              <JTM>

                        ;
                        ;   STANDARD HARDWARE FLOPPY DISK WRITE ROUTINE
                        ;
```

APPENDIX 16

BIOS MODIFICATIONS, VIRTUAL DISK DRIVERS

```
                        ;
                                IF    DISK11      ;IF VIRTUAL FLOPPY SYSTEM        <JTM>

                        ;PDP-11 ASYNCHRONOUS INTERFACE READ ROUTINE                 <JTM>
                        ;
                        ;       READ A SECTOR
                        ;
7E17 3E01               READ11: MVI   A,1           ;SET PDP11 COMMAND TO +1 FOR READ<JTM>
7E19 CD657E                     CALL  TS11          ;SEND TRACK AND SECTOR TO PDP11 <JTM>
7E1C C0                         RNZ                 ;NON-ZERO RETURN MEANS BAD SECTOR<JTM>
7E1D C5                         PUSH  B             ;SAVE B-C                        <JTM>
7E1E 06FF                       MVI   B,0FFH        ;SET COUNT FOR 128 CHAR.         <JTM>
7E20 2A817E                     LHLD  DMAADD        ;H-L -> STANDARD BUFFER          <JTM>
                        ;
7E23 CD2E7E             RSEC11: CALL  CRD11         ;READ A CHARACTER FROM PDP-11    <JTM>
7E26 23                         INX   H             ;POINT H-L TO NEXT BUFFER BYTE   <JTM>
7E27 05                         DCR   B             ;READ 128 CHARS YET ?            <JTM>
7E28 C2237E                     JNZ   RSEC11        ;IF NOT CONTINUE READING         <JTM>

7E2B C1                         POP   B             ;WHEN WE'RE DONE RESTORE B-C     <JTM>
7E2C AF                         XRA   A             ;  MAKE RETURN GOOD              <JTM>
7E2D C9                         RET                 ;    AND GO BACK TO CALLER       <JTM>
                        ;
                        ;       READ A SINGLE CHARACTER FROM PDP-11
                        ;
7E2E 3E80               CRD11:  MVI   A,80H         ;SET HI BIT ON CONTROL PORT (3)  <JTM>
7E30 D303                       OUT   3             ; TO SELECT DATA PORT            <JTM>
7E32 DB01                       IN    1             ;READ THE DATA PORT              <JTM>
7E34 77                         MOV   M,A           ;STORE BYTE IN MEMORY BUFFER     <JTM>
7E35 CD5B7E                     CALL  RDYWT         ;WAIT FOR TRANSFER COMPLETION    <JTM>
7E38 C9                         RET                 ;  AND GO BACK TO CALLER         <JTM>
                        ;
                        ;PDP-11 ASYNCHRONOUS INTERFACE WRITE ROUTINE                <JTM>
                        ;
                        ;       WRITE ONE SECTOR                                    <JTM>
                        ;
7E39 3EFF               WRIT11: MVI   A,0FFH        ;SET PDP11 COMMAND TO -1 (WRITE)<JTM>
7E3B CD657E                     CALL  TS11          ;SEND TRACK AND SECTOR TO PDP11  <JTM>
7E3E C0                         RNZ                 ;NON-ZERO RETURN MEANS BAD SECT  <JTM>
7E3F C5                         PUSH  B             ;SAVE B-C                        <JTM>
7E40 06FF                       MVI   B,0FFH        ;SET COUNT FOR 128 BYTES         <JTM>
7E42 2A817E                     LHLD  DMAADD        ;USE STANDARD BUFFER <- H-L      <JTM>
7E45 CD507E             WSEC11: CALL  CRW11         ;WRITE A CHARACTER TO PDP11      <JTM>
7E48 23                         INX   H             ;H-L POINTS TO NEXT BUFFER BYTE  <JTM>
7E49 05                         DCR   B             ;WRITTEN 128 CHARS YET ?         <JTM>
7E4A C2457E                     JNZ   WSEC11        ;IF NOT CONTINUE READING         <JTM>

7E4D C1                         POP   B             ;WHEN WRITE COMPLETE RESTORE B-C<JTM>
7E4E AF                         XRA   A             ;MAKE RETURN CODE GOOD           <JTM>
7E4F C9                         RET                 ;BACK TO CALLER                  <JTM>
                        ;
                        ;       WRITE A SINGLE CHARACTER TO PDP-11                  <JTM>
                        ;
7E50 3E80               CWR11:  MVI   A,80H         ;SET HI BIT ON CONTROL PORT      <JTM>
7E52 D303                       OUT   3             ; TO SELECT DATA PORT            <JTM>
7E54 7E                         MOV   A,M           ;GET BYTE FROM BUFFER            <JTM>
```

```
7E55 D301               OUT  1          ;SEND IT TO PDP-11                  <JTM>
7E57 CD5B7E             CALL RDYWT      ;WAIT FOR TRANSFER COMPLETION       <JTM>
7E5A C9                 RET             ; THEN BACK TO CALLER               <JTM>
                ;
                ; WAIT FOR HI BIT OF PDP-11 ASYNCH INTERFACE STATUS        <JTM>
                ; REGISTER TO BE SET BY PDP-11 SOFTWARE. THIS INDICATES    <JTM>
                ; THAT THE PDP-11 IS READY FOR ANOTHER DATA TRANSFER       <JTM>
                ;
7E5B AF         RDYWT:  XRA  A          ;PUT 0 IN INTERFACE CONTROL PORT<JTM>
7E5C D303               OUT  3          ;  TO SELECT STATUS PORT            <JTM>
7E5E DB01               IN   1          ;READ STATUS PORT                   <JTM>
7E60 B7                 ORA  A          ;SET FLAGS.  SIGN BIT IS SET        <JTM>
7E61 F25B7E             JP   RDYWT      ;   WHEN TRANSFER IS DONE           <JTM>
7E64 C9                 RET             ;RETURN TO CALLER                   <JTM>
                ;
                ;       SEND TRACK AND SECTOR NUMBERS TO PDP-11 SOFTWARE <JTM>
                ;
7E65 21917E     TS11:   LXI  H,TEMP     ;USE TEMP STORAGE FOR COMMAND       <JTM>
7E68 77                 MOV  M,A        ;PUT COMMAND IN MEMORY              <JTM>
7E69 CD507E             CALL CWR11      ;SEND TO PDP11                      <JTM>
7E6C 217F7E             LXI  H,TRK      ;POINT H-L TO TRACK NUMBER          <JTM>
7E6F CD507E             CALL CWR11      ;SEND TRACK TO PDP11                <JTM>
7E72 23                 INX  H          ;POINT TO SECTOR NUMBER             <JTM>
7E73 CD507E             CALL CWR11      ;AND SEND IT TO PDP11               <JTM>
7E76 21917E             LXI  H,TEMP     ;USE TEMP STORAGE FOR RETURN CODE<JTM>
7E79 CD2E7E             CALL CRD11      ;PDP-11 RETURNS 0 IF GOOD           <JTM>
7E7C AF                 XRA  A          ;MAKE ZERO FOR COMPARE              <JTM>
7E7D BE                 CMP  M          ;SET FLAGS                          <JTM>
7E7E C9                 RET             ;LET CALLER LOOK AT COND CODE       <JTM>

                        ENDIF                                               <JTM>
                ;
                ;NOTE:  AS THERE ARE ONLY NINE SECTORS
                ;AVAILABLE FOR CBIOS ON THE SECOND SYSTEM TRACK (1),
                ;THE LAST ADDRESS BEFORE THIS POINT SHOULD BE NO
                ;GREATER THAN THE CBIOS STARTING ADDRESS + 047F (HEX).
                ;THIS WILL NORMALLY BE XE7F (HEX).
                ;
```

VITA

James T. McCrae was born in Canton, Illinois on March 6, 1952. He attended elementary schools in Farmington, Illinois and was graduated from Farmington East High School in May 1970. He entered The University of Illinois at Champaign-Urbana in August 1970 and received a Bachelor of Science degree with a major in Psychology in June 1974.

After working in Peoria, Illinois, he started working full-time for The University of Tennessee, Knoxville in December 1974 and entered graduate school there in March 1975. In April 1976 he started working full-time as an operator for The University of Tennessee Computer Center. In May 1979 he began working as a full-time process control programmer for Computer Concepts Corporation of Knoxville.

He began work with the Communications Satellite Corporation of Palo Alto, California in December 1980. He will receive the Master of Science degree with a major in Computer Science from The University of Tennessee in August 1981.