120

/An Implementation of the Kermit Protocol Using the Edison System/

by

Terry A. Scott

B. S. Iowa State University, 1964

Ph. D. University Of Wyoming, 1972

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

Approved by

Rick McBride

Major Professor

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

ACKNOWLEDGEMENTS

The author would like to express sincere thanks to Professor Richard McBride for suggesting this problem and for his help and encouragement in its completion.

In addition, thanks go to Bill McDaniel, Harvard Townsend, and Earl Harris. Without their help, the completion of this project and report would have been more difficult.

Chapter 1: Introduction

1.1 The project.

The Edison operating system and program development environment [Bri 1982] was used to implement the Kermit protocol [Cru 1984A][Cru 1984B] for file transfer via the serial port on the PC microcomputer. This was done by changing the original operating system so that there are new operations that can be done. These operations allow files to be sent and received between the PC microcomputer and the Kermit running on a Unix-based mini-computer.

1.2 Why the project was done.

The Edison operating system and programming environment, hereafter called the Edison System, is relatively simple and can be learned in a very short time. The Edison language that is the medium for program development is also relatively easy to learn and supports facilities for creating concurrent programs. The simplicity and the features of this programming language make it an ideal tool for learning and writing concurrent programs. During program development it would be helpful to use an editor other than the rather primitive one included in the Edison System. The transfer of files between the micro and another computer with more sophisticated editing is a real advantage.

Eventually the Kansas State University Computer Science Department would like to be able to run Edison programs and even the Edison operating system on a minicomputer in the department. A file transfer system that allows text and object files to be moved to Unix would facilitate this project considerably.

In addition, the project was a real challenge, and a great deal was learned in arriving at a solution.

1.3 The Edison System.

The Edison System was written by Per Brinch Hansen. The system was written to be run either on the PDP-11 computer or the IBM PC microcomputer. He designed the operating system and environment with the two main goals of simplicity and the ability for a programmer to write nontrivial programs with a minimum of difficulty [Bri 1982]. He mentions several design rules he used to accomplish these goals: keep the system small, use the same language throughout, hide machine details, and use uniform interfaces to the terminal, printer. and the disks [Bri 1982].

1.4 The Edison language.

Per Brinch Hansen designed and implemented the Edison language. This language was used to implement all of the Edison System except for the kernel which is written in an assembly language called Alva. Alva is a subset of PDP-11 assembly code with slight changes in syntax. The changes in syntax generally make the commands more meaningful [Bri 1982]. For example the PDP-11 assembly command BNE becomes ifnotequal in Alva. Brinch Hansen chose to make the Edison language somewhat like Pascal[Bri 1982], but he has made judicious additions, subtractions, and changes. He made these decisions based on making the overall system as simple, compact, and unambiguous as possible [Bri 1982].

1.5 Kermit protocol.

Kermit [Cru 1984A] [Cru 1984B] is a sliding window protocol of size one. It uses a single character checksum to determine if errors have occurred in transmission. Data and control information are clumped together in units called packets.

A sender transmits a packet with a sequence number and a checksum. When the receiver has received the packet, it checks the sequence number and checksum value. If it agrees with both of the values, it sends a positive acknowledgment(ack) back to the sender. If there is disagreement on the checksum it sends a negative acknowledgment(nak). If the checksum is okay, but the sequence number is one less than it should be, then the previous packet is acked. If the sequence number is neither the current value nor the previous value, then a nak is sent. If the sender receives an ack, it sends the next packet, otherwise it will timeout after some specified time and it will retransmit the same packet. If no headway is made in some predefined time or no response is received by either of the sides, then the protocol times out, and the file transfer must be started again.

1.6 The report organization.

This report may be helpful to persons that want to: develop a program on the Edison System, transfer files between an Edison System and a Unix system, learn about the Kermit protocol. read about the Edison-Kermit system, or modify the Edison operating system.

For readers interested in program development, chapters 2, 3, and appendix B should be helpful. If you are only interested in transferring files then you should read chapter 5 and appendix A. Chapter 4 and appendix C should be helpful in learning more about the Kermit protocol. Information about the Edison-Kermit system is contained in chapters 5 and 6 and appendices D and E. Finally chapters 2, 3, and appendix B are useful in learning how to change the Edison operating system.

Chapter 2. Edison Language.

2.1 Introduction.

In this chapter the Edison and Pascal languages will be compared. Brinch Hansen wrote the Edison language for the Edison System. The operating system and all utility programs were written in this language except for the kernel. The kernel is written in Alva, which is an assembly language similar to PDP-11 assembly language [Bri 1982].

As was discussed in chapter one, Brinch Hansen used the design rules of simplicity and compactness in designing the overall system [Bri 1982]. In addition, the Edison language was written specifically for implementing operating systems. The Edison language is similar to the Pascal language, but Brinch Hansen has made additions and deletions from Pascal to keep the language simple, compact, and oriented toward operating system applications.

The remainder of this chapter will discuss Edison from the viewpoint of syntactic differences with Pascal, semantic deletions from Pascal, and semantic additions to Pascal. All keywords in Pascal and Edison will be capitalized throughout the rest of this chapter.

2.2 Syntax Changes From Pascal

The following table shows the Standard Pascal [Wir 1971] syntax in the left column, and in the right column the equivalent Edison syntax. The angle brackets ,i.e., <, > around a word signify a non-terminal. The word prefix used in the parameter list of a program is a set of constants and procedures that are defined in the operating system and passed to a program in the program parameter list.

```
Standard Pascal                 Equivalent Edison
Syntax Examples                 Syntax Examples


PROGRAM <identifier>            PROC <identifier>
         (<files>)                   (<prefix>)

INTEGER                         INT

BOOLEAN                         BOOL

ORD                             INT

CHR                             CHAR

TYPE sector =
  ARRAY[1..256] OF INTEGER      ARRAY sector[1:256](INT)

TYPE RECORD =                   RECORD item(number, price,
        item                                weight: INT)
          number: INTEGER;
          price: INTEGER;
          weight: INTEGER
        END

TYPE daytype =
           (mon,tues,wed)       ENUM daytype(mon,tues,wed)

PROCEDURE                       PROC

FUNCTION                        PROC

IF <condition> THEN             IF <condition> DO
  <stmt>                          <stmt>
                                END
```

```
IF <condition> THEN            IF <condition> DO
BEGIN                            <stmts>
  <stmts>                      END
END

WHILE <condition> DO           WHILE <condition> DO
BEGIN                            <stmts>
  <stmts>                      END
END

{a comment}                    "a comment"

In the body of a function a value is assigned to the
function in the following way.

<function name> := <value>   VAL <function name> :=
                                                <value>

A program is terminated in the following way.

END.                           END
```

Figure 2.1: Pascal and Edison Syntax Comparison.

The other main syntax change involves the use of the semicolon. In Edison the semicolon is used to separate statements in a statement list. However no semicolon should be placed in front of words such as VAR, CONST, BEGIN, PROC, ARRAY, RECORD, ELSE, MODULE, or END. Pascal places a null statement after a semicolon that is placed before a reserved word such as END. This allows the programmer to be somewhat sloppy about semicolon placement. Edison is not forgiving of this kind of error, and you must place all semicolons correctly before a program will compile.

2.3 Deletions From Pascal.

This section will discuss the semantic deletions from Pascal. It will be organized by covering in order the deletions from: flow of control constructs, data types, and operations on data.

The FOR statement, REPEAT-UNTIL statement, and CASE statement have been left out of Edison. Because the WHILE, IF-DO, and IF-DO-ELSE-DO statements allow a programmer to do everything that is necessary, these other flow of control statements were not included.

The REAL, POINTER. and subrange types have been deleted from Edison. These types are not generally needed in writing operating system type programs.

The Edison language has no variant records which is in keeping with simplicity, compactness, and the non-necessity of this concept in programming. Edison has no nameless types which is sometimes referred to as implicit typing in Pascal. If a user-defined type is needed, it must be declared as a type, and then that name is used when declaring a variable of that type.

The WITH statement in Pascal can save some time in entering a program. but it can make a program more difficult to read, particularly if the WITH statements are nested. Because it is a luxury, and it can make a program more difficult to read, the with statement was not included in Edison.

The SUCC and PRED functions have been left out of Edison. There are no standard input/output routines ,e.g., WRITELN and READ or file operations in Edison as there are in Pascal. All input/output routines and file operations are defined in the operating system, and are accessed by a program through the prefix.

2.4 Additions To Pascal.

Edison has some very nice additions that in some ways make it an easier language to work with than Pascal. The additions will be discussed under the general topics of ARRAY and RECORD constructors, modules, retyping, flow of control constructs, and concurrency.

In Pascal it is necessary to have a separate statement for every assignment made to an ARRAY element and RECORD field. However, in Edison it is possible to assign values to every element of an ARRAY and every field of a RECORD with a single statement. These constructs are called ARRAY and RECORD constructors. The example below demonstrates the syntax of a constructor.

```
ARRAY lengths[1:12](INT)
VAR month_lengths : lengths
BEGIN
   month_lengths := lengths(31, 28, 31, 30, 31, 30,
                            31, 31, 30, 31, 30, 31);
```

The assignment statement in this example places a 31 in month_lengths[1], a 28 in month_lengths[2], etc.

Edison has a module structure which is a way of placing related procedures, functions, and data types together. Not only does this make the code more readable, but it serves to encapsulate the data types and the operations that can be performed on them. This concept is sometimes called an abstract data type. The module concept modifies the scope of the data types, procedures, and functions inside the module. Only those named types with an asterisk in front of them may be exported to the surrounding scope.

There is a problem with this concept as implemented in Edison. If two modules in the same scope have exported identifiers that have the same name, then there will be an unresolved name ambiguity. This makes it necessary to keep track of identifier names in different modules with the same scope. One nice aspect of an Edison module is that each module has an initialization portion which is performed when the module is started up. This is similar to a class or monitor type in Concurrent Pascal [Bri 1977].

The type concept serves very well to make certain that a programmer does not do anything foolish when he is passing parameters or performing operations on a data type. However, sometimes it is important to relax type checking. An example of this is when a text file is to be retrieved from the disk. The information is stored on the disk as integers, but if retrieved as integers, they would have to be converted to characters. It would be far easier to read the text file from the disk as characters and no conversions would then be necessary. To accomplish this Brinch Hansen introduced the concept of data retyping [Bri 1982]. The following example should explain this idea.

```
ARRAY sector[1:256](int)
ARRAY page[1:512](char)
VAR block: page
BEGIN
  readsector(block:sector)
```

The variable block is declared to be of type page. The definition of readsector has a parameter of type sector, and therefore the variable block can not be used directly at the call. The solution is to retype block to be of type sector when the readsector procedure is called. The reason for the apparent different memory sizes is because the integer type in Edison is two bytes long. This illustrates an important point about retyping; retyping can only be done between two types that require the same amount of memory.

The two main flow of control constructs in Edison have some added features which make them more useful. The IF-DO-ELSE-DO requires a condition after the else so that it is similar to the ELSE-IF in Pascal. An example will illustrate how this is done.

```
IF sex = 'f' DO
  num females := num_females + 1
ELSE sex = 'm' DO
  num males := num_males + 1
ELSE TRUE DO
  write_line(display,
            line('  Error, sex must be m or f.'))
```

The write_line statement is Edison's way of displaying a line of text on the screen.

The other main flow of control mechanism is the while loop. It is possible to write an ordinary while loop that looks very much like Pascal, however else clauses can be added to the WHILE loop. An example should illustrate this.

```
accept(petkind);
WHILE petkind = 'd' DO
  num dogs := num_dogs + 1;
  accept(petkind)
ELSE petkind = 'c' DO
  num cats := num cats + 1;
  accept(petkind)
ELSE petkind = 'b' DO
  num birds := num_birds + 1;
  accept(petkind)
END
```

The accept statement takes a value from the keyboard and places it in the memory location of petkind. As long as the value read from the keyboard is a 'd', 'c', or 'b', the statements under the clause with the true condition will continue to be done. If it is any other value, it will quit the loop and begin executing on the statement following END.

The SKIP statement has been included in Edison as a no operation statement. It can be used anytime that it is syntactically necessary for a statement, but nothing is done by the statement. The only time this statement has been used in this project is between a BEGIN and END in the initialization portion of a module.

The Edison language has semantic constructs for doing concurrency. To execute several statements concurrently it is necessary to use the following construct, where sl1, sl2,

..., sln stand for statement lists 1, 2, ..., n.

```
COBEGIN 1 DO
  <sl1>
ALSO 2 DO
  <sl2>
   ...
ALSO n DO
  <sln>
END
```

Of course the IBM PC style computers have only one cpu, so "concurrently" just means that the statements can potentially be executed in any order.

If the statement lists in the previous example operate on shared data. then it is necessary that there be no context switch that would allow improper evaluation of the data. To avoid this there must be a way of synchronizing the concurrent statement lists. A mechanism for accomplishing this synchronization, called a conditional critical region was first published by Tony Hoare [Hoa 1972]. Process synchronization using conditional critical regions in Edison is accomplished by a when statement, which is illustrated below.

```
WHEN <condition1> DO
  <sl1>
ELSE <condition2> DO
  <sl2>
  . . .
ELSE <condition3> DO
  <sl3>
END
```

There are two separate parts to the execution of a WHEN statement. During the synchronization phase a process is delayed until there are no processes executing a critical phase of a WHEN statement. Once the critical phase of all WHEN statements is clear, a process is allowed into the critical phase. The conditions of the WHEN and ELSE clauses are checked one at a time in the order in which they are written. If a condition is true, then the statement list of that clause is executed and the WHEN statement is terminated. If none of the conditions are true, then the process returns to the synchronizing phase.

Chapter 3. Edison System and Program Development Environment.

3.1 Introduction

This chapter will discuss how to use the Edison System and the steps in developing an Edison program. First, all of the Edison operating system commands will be given along with a brief description of what each does. Next the auxiliary programs that are useful in developing an Edison program will be discussed, and finally the steps that are needed in developing a program and in changing the kernel and operating system will be listed.

3.2 Edison Operating System Commands.

In the following paragraphs a list of all twelve operating system commands will be given along with a brief description of what each of them does and an explanation of its usefulness. Further descriptions of these commands are contained in chapter 5 of reference [Bri 1982] in the bibliography.

List. The list command displays names, their sizes, and protected/unprotected status of all files on a floppy disk. This command is indispensable.

Formatdisk. The formatdisk command readies a disk for receiving files. This command must be performed on a disk before any information can be placed on the disk. This means that none of the system commands can be used on a disk until the formatdisk command is performed.

Delete. This command deletes a file from the file catalog and frees the disk space occupied by that file. The delete command only works on unprotected files. To unprotect a file, use the unprotect command listed below. Obviously the delete command is very important.

Protect. This command protects a file so that it can not be overwritten, deleted, or renamed. If a file is important it should be protected so that it will not be accidentally lost.

Unprotect. This command changes a file's status from protected to unprotected. Once the file is unprotected it can be overwritten, deleted, or renamed. If a file is protected it obviously is important to be able to unprotect it so that it can be removed from the disk when necessary.

Copy. The copy command copies a file into an empty file. This can be used to make a backup copy of a file, and it also allows the prefix to be copied into a new file prior to starting a new program. Being able to copy the prefix saves about 50 lines of typing for each new program that is created. If the name of the file to be copied is in the file catalog of both disks, the file on disk 0 will be the one that is copied.

Backup. The backup command copies a disk's contents, sector by sector to another disk. This is useful for backing up a complete disk because it copies the operating system and kernel in addition to the files in the catalog. Backups should be made of all Edison disks to prevent accidental erasure or loss due to a disk failure.

Rename. This command allows a file's name to be changed if the file is not protected. To unprotect a file use the unprotect command explained above. There are times that it would be helpful to rename a file by choosing a more meaningful name. It can also be used to prevent a name ambiguity between the two disks or an Edison file and a Unix file.

Newsystem. If a change is made in the operating system program, the changed code is compiled to obtain object code, and then the newsystem command is used to install this new object code. This command is only used if changes are to be made to the operating system. If the name of the new operating system file exists in the file catalogs of both disks, the file on disk 0 will be the one installed as the operating system.

Newkernel. If a change is made in the kernel program, the changed code is assembled to obtain object code and then the newkernel command is used to install this new object code. This command is only used if changes are to be made to the kernel. If the name of the new kernel file exists in the file catalogs of both disks, the file on disk 0 will be the one installed as the new kernel.

Insert. The insert command can be used when a new disk or disks are placed in the disk drives. This command updates the file catalogs in memory without the necessity of rebooting the system.

Create. The create command is used to create a new, unprotected, and empty file. This command should be of limited usefulness to system users. If a file is needed, it can be created by typing edit and a new file name, or by using the copy command to copy the prefix command into a new file.

3.3 Edison System auxiliary Programs.

This section will discuss the auxiliary programs that are outside the operating system, but are necessary in developing a program. The two programs called format and underline are not needed in program development and they will not be discussed here. If you wish information on these two programs or further information on the programs that will be discussed below, see chapter 5 in reference [Bri 1982].

Whenever an auxiliary program is used ,e.g., edit and compile, a file name will be requested. If the file name given is in the file catalogs of both floppy disks, the Edison operating system will check the file catalog of disk 0 first, and use that file. If you want to use an auxiliary program on a file on disk 1, be certain no file by that name exists on disk 0.

The following are programs which are necessary or helpful in program development: edit, compile, assemble, print, cut, and paste.

Edit. The edit command allows new lines to be added to a file, changes to be made to a file, and lines to be deleted from a file. The user should be aware of several things. When the user leaves the editor the program will ask if the changes are to be saved. If that is true, then you must type exactly the word yes. If you type anything other than the exact word yes, then the changes will not be written to the disk.

The user should check before he enters the editor to determine the size of the file to be edited. There must be at least that many free pages on the disk where the file is to be saved. For example, if a file is 50 pages and it resides on disk 1 and will be saved on disk 1, then there must be at least 50 free pages on that disk, or the file will not be saved.

A further check should be made on ambiguous names. This can cause trouble when a file on one disk is edited and it is to be saved on the other disk. If a file of the same name as the edited file already exists on the disk, then there will be a name conflict, and the file will not be saved.

With the ability to transfer files back and forth between Unix and the Edison System, the workings of the Edison System editor may not be as important. Only for the case of a file with a small number of required changes should it be advantageous to use the Edison System editor. For all other situations it will probably be easier and faster to edit the file on Unix and then transfer it back to the microcomputer with the Edison-

Kermit system.

Compile. In the following description, the expression <name> is a nonterminal and any legal name may be used in its place. Brinch Hansen's convention of calling source files <name>text and the compiled code <name> was used [Bri 1982]. This helps to keep straight which files are executable and which executable files go with which source files.

One difficulty that was encountered in using the compiler was exceeding the name table size. A list of possible compiler failures can be found on pages 139-140 of reference [Bri 1982] in the bibliography.

Assemble. The only time a user would need this command is if it is necessary to make changes in the kernel of the Edison System. The kernel is in two parts: kerneltext1 and kerneltext2. In the case of this project, only changes were made to the kerneltext1, but the assemble command requires that both parts be assembled even though changes were made to only one of the files.

Print. For the print command to work, a printer must be attached to the parallel port of the microcomputer. This program may not be as useful now that a file can be transferred to Unix and printed on a high speed printer.

Cut. The cut program allows a part of a file to be copied to another file.

Paste. The paste program allows a file to be appended to the end of another file.

The cut and paste programs can be used to move pieces of a file to another location in the file or to take a piece from an existing file and place it in a new file. This can save considerably on typing. These two programs are probably not as useful now that it is possible to transfer a file to Unix, edit the file with vi, and then transfer the file back to Edison.

3.4 Developing an Edison Program.

This section will discuss the steps in developing an Edison program and how to change the operating system and kernel.

Initially assume that the program development will be done with no file transfer to Unix. In that case the development would follow these steps.

1. Place two Edison disks in the microcomputer and switch on the computer.

2. Make certain the prefix file is on one of the disks, and

copy the prefix file into a file with text as the suffix for the file name.

3. Edit this file and begin adding your program code below the prefix.

4. Once the program is the way you want it, hit the F6 key and type yes when the prompt asks if you want to save the changes.

5. Compile the program and name the object file the same name as the source file without the text suffix.

6. If there are compile errors, edit the text file. Once in the editor type s, and the editor will display the compile errors. Repeat steps 4, 5, and 6 until all compile errors are removed. Be aware that the compiler may not report all errors after a compilation. If an error is made in the block structure of the language, the compiler may become confused and may not check part of the program.

7. Once all compile errors have been removed, run the program by typing the object file name to the Edison System prompt ,e.g., Command = .

8. If there are execution errors, the Edison System will give a line number where the error occurred. Edit the text file, locate where the error occurred, fix the error, and recompile the program. Run the program as in

number 7 and continue until the execution errors are removed.

If there are substantial changes to be made to the file, it should be preferable to do the editing on Unix. If you decide on this route, then transfer the prefix over to Unix. Consult appendix A of this report to see how this is done. Once the prefix is transferred, you should make a copy of it so that this transfer need only be made once.

Use vi or the editor of your choice to construct the program you want. Do not forget that the prefix should be at the start of your program. Once the file is the way you want it, then transfer the file back to a floppy disk on the microcomputer, and proceed starting at step 5 above.

3.5 Changing the Edison Kernel.

It is important that a backup system disk always be available. If the kernel is changed incorrectly, it may very well make the Edison operating system unuseable. To change the kernel of the Edison System, proceed with the following steps.

1. Boot up the Edison operating system with the disk containing the kerneltext files in drive 0 or 1.

2. Determine which of the kerneltext files you must change.

3. Edit the proper file and make the needed changes.

4. Once the kernel is the way you want it, hit the F6 key and type yes when the prompt asks if you want to save the changes.

5. Assemble both parts of the kernel and come up with a unique name for the object file.

6. Once the kernel is assembled with no assembly errors, use the newkernel command to place the kernel on pages 0 through 6 where it will replace the old kernel. The newkernel procedure will ask which disk is to receive the kernel. You may type either 0 or 1, but if you choose disk 0, make certain that a backup system disk is available, because an incorrect kernel will very likely make the operating system unuseable.

7. To test the new kernel you should make certain that the disk with the new kernel program has an operating system in place. Place the disk with the new kernel in drive 0 and reboot the operating system by simultaneously holding down the control, alt, and delete keys.

8. If the changes you made are incorrect then you must start over at step three.

To avoid the Edison editor, you may at steps 3 and 4 transfer the kerneltext file to Unix, make the needed changes, and then transfer the file back to the Edison System.

3.6 Changing the Edison Operating System.

It is important that a backup system disk always be available. If the operating system is changed incorrectly, it may very well make the Edison operating system unuseable. To change the operating system of the Edison System, proceed with the following steps.

1. Boot up the Edison operating system with the disk containing the system text file in drive 0 or 1.

2. Edit the operating system file and make the needed changes.

3. Once the program is the way you want it, hit the F6 key and type yes when the prompt asks if you want to save the changes.

4. Compile the program and come up with a unique name for the object file.

5. Once the operating system is compiled with no compile errors, use the newsystem command to place the new system on pages 7 through 24 where it will replace the old operating system. The newsystem procedure will ask which disk is to receive the operating system. You may type either 0 or 1, but if you choose disk 0, make certain that a backup system disk is available, because an incorrect operating system will very likely make the operating system unuseable.

6. To test the new operating system you should make certain that the disk with the new operating system has a kernel in place. Place the disk with the new operating system in drive 0 and reboot the operating system by simultaneously holding down the control, alt, and delete keys.

7. If the changes you made are incorrect then you must start over at step two.

To avoid the Edison editor you may replace steps 3 and 4 by transferring the operating system text file to Unix, make the needed changes, and then transfer the file back to the Edison System.

Chapter 4: Kermit Protocol

4.1 Introduction.

This chapter will discuss the Kermit protocol [Cru 1984A][Cru 1984B]. All information is sent in lumps of data and control information. These lumps are called packets. In the terminology of network theory, the word frame is more correct, but the Kermit implementers use the word packet. First, the form of the packets and what values are allowed in each field will be described, then the order in which the packets are sent will be discussed, and finally the differences between the handling of text and binary files.

4.2 Packets.

4.2.1 General Packet Structure.

Figure 4.1 shows the generalized format of a packet.

SOH	LEN	SEQ	TYPE	DATA	CHECK

Figure 4.1: Kermit Packet Fields.

Because Kermit should be a completely portable protocol, it should send only printable characters with the exception of the control A character used at the start of each packet.

This is necessary because some control characters may cause problems for computers at one or the other end.

4.2.2 Header Field.

The SOH field is start of header. Traditionally SOH has been ASCII value 1, which is a control A, and this character is reserved on most computers for just this purpose [Cru 1984A].

4.2.3 Length Field.

The LEN field stands for the length of the packet. It is always a single printable character. The character is obtained by counting all the characters in the packet after the length field, adding 32 to the value, and converting the integer value to its equivalent ASCII character. In ASCII all the characters between and including 32 to 126 are printable. By taking 126 and subtracting 32 and adding 2 for the two field characters at the beginning of the packet, the value 96 is obtained. This should be the maximum total length of a packet. If the 5 control characters are subtracted from 96 this leaves a maximum of 91 characters in the data field.

4.2.4 Sequence Number Field.

The SEQ field stands for the packet sequence number. The sequence number is used by both sender and receiver to know which packet has just been received. If a positive acknowledgment were lost, then when a packet is resent, the receiver might think it is a new packet and write it to the file. However the sequence number allows the receiver to know that the packet has already been received and the packet can be discarded. This field is a single printable character and starts at ASCII character 32, the blank character. This is considered sequence number 0. Each succeeding sequence number is just the next ASCII character up to and including character 95. After 95 the sequence starts over.

4.2.5 Type Field.

The TYPE field is also a single printable character. There are several different types of packets that are sent. The following is a list of the type of packets that were implemented on the Edison-Kermit system.

S "Send Initialization" Packet. This is transmitted by the sender and prepares the receiver to start receiving. The data field contains information about the sender's parameters for such things as the timeout period. This topic will be discussed later in section 4.2.7.

F "File Header" Packet. This contains the file name in the data portion of the packet.

D "Data" Packet. The data that are to be transferred will be sent in the data portion of the packet.

Sometimes it is necessary to send some control characters such as a new line or carriage return character. These are sent with a # character prefix and the sixth bit is flipped. This has the effect of adding 64 to all characters with ASCII values less than 32 and subtracting 64 from ASCII character 127. A binary file may require that the high bit be on. Unix does not use this bit for parity, therefore it is okay for bit seven to be on. The treatment of bits zero through six ignores the value of bit seven.

Z "End of File" Packet. This indicates the end of the file. The data field will always be empty.

B "End of Transmission" Packet. This indicates there are no more files to be sent. The data field will always be empty.

Y "Positive Acknowledgment" Packet. This is sent by the receiver when it receives a correct packet. The data field will be empty except when it responds to the send initialization packet. In that case it will contain the receiver's values for such things as the timeout period. This will be discussed later in section 4.2.7.

N "Negative Acknowledgment" Packet. This is sent by the receiver when it receives an incorrect packet.

The data field will always be empty.

4.2.6 Checksum Field.

Finally the CHECK field is a single printable character checksum. Making this character printable avoids control character problems with the computer, and it also makes it easy for a system developer to analyze a packet and see that the checksum value is correct. The CHECK field character is obtained by summing the ASCII values of all characters in a packet with the exception of the header and the check fields. This value is converted into binary and only the low order byte is kept. Bits 7 and 6 of this byte are moved to bit locations 1 and 0 respectively and added to bits 0 through 5. This sum is converted to a decimal value, 32 is added to this number, and the result is converted to an ASCII character. The largest value that can be represented by 6 bits is 63, and counting the value 0 there is a total of 64 possibilities. If 0 is represented by ASCII character 32, the first ASCII printable character, then the largest ASCII value that will occur will be 95, and this is a printable character.

4.2.7 Initialization Data Fields.

The "Send Initialization" packet and the receiver's acknowledgment of that packet contain values in the data field for the initial values of the sender's and receiver's protocol parameters [Cru 1984B]. The data field for these packets is shown in Figure 4.2.

```
 ______________________________________________________
:    :    :    :    :    :    :    :    :    :    :
:MAXL:TIME:NPAD:PADC:EOL :QCTL:QBIN:CHKT:REPT:RES :
:    :    :    :    :    :    :    :    :    :    :
:____:____:____:____:____:____:____:____:____:____:
```

Figure 4.2: Initialization Data Fields.

The terms computer1 and computer2 will be used in the following descriptions. The computer1 refers to the sender of the packet and the computer2 refers to the receiver of the packet. In the case of numeric values, the values are changed to printable ASCII character by taking the value, adding 32, and converting that value to an ASCII character. These parameters have default values, and the default value will be used if no parameter value is sent.

MAXL	This field gives the maximum length of packet that computer1 wants to receive. Computer2 responds with the maximum it wants computer1 to send.
TIME	The amount of time in seconds that computer1 wants before it is timed out by computer2.
NPAD	The number of padding characters computer1 wants before each packet that is sent to it by computer2.
PADC	The padding character computer1 wants sent to it before each packet.

EOL	The character computer1 wants sent to it to terminate a packet if any.
QCTL	The printable character that computer1 expects to prefix a control character.
QBIN	The printable character that computer1 expects to prefix a byte with bit seven on. This is unnecessary in Unix since bit seven can be sent in a normal manner.
CHKT	The type of error checking that computer1 will use. The number 1 is the simplest, and it uses a one character checksum. The two other methods are chosen by using 2 or 3 in this field. If computer2 agrees with computer1 on the type of error checking, then that method will be used. If there is disagreement, then error checking method 1 will be used.
REPT	The prefix character computer1 will use to indicate a repeated character. Any character can be used except for blank.
RES	This is a reserved area to be used by the individual user and for future use by the Kermit developers.

4.3 The order of sending packets.

As mentioned in the introduction of this report, Kermit is a sliding window protocol of size one. Each packet that is sent is checked by the receiver for the proper sequence number and checksum. If the checksum and sequence number are the expected values, then the receiver sends a "Positive Acknowledgment" for that packet. If the checksum does not agree with the calculated value, then a "Negative Acknowledgment" is sent. If the checksum is correct and the sequence number is one less than the expected value then a "Positive Acknowledgment" packet is sent for the previous frame. If the checksum is correct, but the sequence number is not the expected value or one less than the expected value, then a "Negative Acknowledgment" is sent. The sender always waits for a "Positive Acknowledgment" before it goes on. If it receives a "Negative Acknowledgment" or no acknowledgment within the allowed time, it resends the packet. It will resend a packet several times before giving up. If it gives up, the protocol must be restarted.

The receiver checks each packet it receives. If the checksum, sequence number, packet type are the expected values, then a "Positive Acknowledgment" is sent. If the checksum or packet type are incorrect, then a "Negative Acknowledgment" is sent. If the sequence number is one less than the expected value, then that previous packet is positively acknowledged. Any sequence number other than the expected value or one less than the expected value cause a "Negative Acknowledgment" to be sent.

When the receiver gets the first packet, a "Send Initialization" packet, it sends an acknowledgment for that packet containing the values of the parameters in Figure 4.2 which it will use. At this time the receiver and sender adjust their packet lengths, timeout periods, and other parameters based on the information contained in the data fields of these two packets. The sender now outputs a "File Header" packet. When the packet is received, the receiver opens a file with that name and acks that packet. Once the sender receives the ack, it starts to send "Data" packets, each time waiting until the receiver acks the last packet.

The "Data" packets continue to be transferred until the end of the file is reached. At this point the sender outputs an "End Of File" packet, and the receiver closes the file when it receives the packet. Finally an "End Of Transmission" packet is sent, and the receiver leaves the receive state. If all packets are positively acknowledged, then the protocol gives a message that the file transfer was successful.

4.4 Text and Binary Files

When Kermit is started up with the command kermit r or kermit s <filename>, a text file will be received or sent. This has two effects on the protocol: bit seven is ignored, and a carriage return and line feed become only a line feed on the receive and a line feed becomes a carriage return and line feed on a send. If the command is kermit ri or kermit si <filename> then a binary or image file will be received or sent. The protocol will now use bit seven, but no carriage return and line feed conversion will be done.

For an example of Kermit session packets, see Appendix C of this report.

Chapter 5. Edison-Kermit System

5.1 Introduction.

This chapter will discuss the Edison-Kermit system. This chapter is divided into the following sections: Edison-Kermit System Commands, Edison-Kermit System Performance, and Aspects of the Edison-Kermit System.

5.2 Edison-Kermit System Commands.

The Edison-Kermit system has 9 commands: delete, list, protect, rename, unprotect, connect, initport, send, and receive. The delete, list, protect, rename, and unprotect commands have been explained in chapter 3 of this report and reference [Bri 1982] in the bibliography.

The commands backup, copy, create, formatdisk, insert, newkernel, and newsystem which were in the original Edison operating system have been deleted from the Edison-Kermit system. In addition no auxiliary programs can be run by the Edison-Kermit system. This was necessary because there was not sufficient room to include all the original commands and code along with the new commands for file transfers. The commands delete, list, protect, rename, and unprotect were included because they take little additional code, and they seemed the most useful commands for a file transfer system.

Initport, connect, send, and receive are the four commands which were added to the original operating system. Each of these commands will be discussed below.

Initport. When this command is given, the system responds with "Type t for default configuration". If the user types t, the serial port on the microcomputer is configured at 1200 baud, no parity, one stop bit, and a byte length of 8 bits. If the user types anything but t, the system will ask "Configuration value =". To decide what value to type, consult Table 5.1 below [Zen 1984].

Baud rate		
	9600 --->	224
	4800 --->	192
	1200 --->	128
	600 --->	96
	300 --->	64
	150 --->	32
	110 --->	0
parity		
	no parity --->	0
	odd parity --->	8
	even parity --->	24
number of stop bits		
	1 --->	0
	2 --->	4
byte length		
	7 bits --->	2
	8 bits --->	3

Table 5.1: Configuration Values.

Add all the values for your choices under each of the four categories. Type this sum for the configuration value. It would be easy to add code which would query the user about what specific configuration was required. The reason this was not done was due to the limited space available for the operating system, since adding this code would make the system too large. Of course some of the extra commands could be left out, but it was decided by the implementer that it was more important to have the commands than to have the option to easily change the port configuration, especially since the default configuration should work in almost all situations.

Connect. This command connects the microcomputer to the serial port so that anything that is typed on the keyboard is sent to the serial port, and anything coming into the serial port is displayed on the screen. This command can be used to make the PC microcomputer into a terminal for the computer attached to the serial port. This command allows the user to login to the Unix operating system, and start Kermit running on the remote computer. The user should hold down the control key and press the q key to return to the microcomputer.

Send. To use the send command you must first connect to the remote computer using the connect command, login to the system, and type kermit r or kermit ri depending on whether you are sending a text or binary file. Return to the microcomputer by pushing the q key while holding down the control key. Once back to the microcomputer type send and press the return key. The Edison-Kermit system will ask which file you want to send. Type the file name to be sent and press the return key. The system will ask whether it is a text or binary file. Type the correct response for this, and hit the return key. The transfer should then proceed with the system recording on the screen how many packets have been sent, and how many packets have been sent more than once. This will continue until the file is sent successfully or unsuccessfully. If the file is sent successfully, a message to that effect will be placed on the screen. If the file transfer is unsuccessful, a message to that effect will be placed on the screen or an error message will be given.

Receive. To use the receive command you must first connect to the remote computer using the connect command, login to the system, and type kermit s or kermit si followed by a file name. Kermit si should be used for all binary files. A text file can be sent with kermit s or kermit si. If kermit s is used, a line feed character in a Unix file will cause a carriage return character and a line feed character to be sent. Since two characters are not needed to terminate a line, it can make an

Edison file longer than necessary. The Edison-Kermit system text file is about 1500 lines long. If it is sent using kermit s, then the file will be about 1500 characters or nearly 3 pages longer than necessary. Sending a file with kermit si eliminates the extra character at the end of each line. Because of this difference, it makes sense to send a text file on Unix with kermit si also. Return to the microcomputer by pressing the q key while holding down the control key. Once back to the microcomputer type receive and press the return key. Edison-Kermit will ask whether it is a text or binary file. Type the correct response for this, and press the return key. Finally the system will ask which drive is to receive the file, and you should type 0 or 1 and press the return key. The transfer should then proceed with the system recording on the screen how many packets have been received correctly, and how many packets have been received more than once. This will continue until the file is received successfully or unsuccessfully. If the file is received successfully, a message to that effect will be placed on the screen. If the file transfer is unsuccessful, a message to that effect will be placed on the screen or an error message will be given.

5.3 Edison-Kermit System Performance.

The system was tested by sending and receiving short files, long files, text files, and binary files. Although it is impossible to test the protocol completely, it does work correctly on all files that have been tried.

My testing checked binary files by observing the word length of a file on a floppy disk and its length after being transferred to Unix and back to Edison-Kermit. In all cases the file length on Unix was twice as long as it was on the floppy disk. This is correct because for a binary file Edison counts two bytes as a word, whereas Unix is counting bytes. Several times working programs were transferred to Unix and back to the microcomputer and then the transferred file was run. In all cases the program behaved as it had before the transfer.

The text files were tested in a similar way by taking a text file on Edison and compiling the file and running the program. Then the text file was transferred to Unix, back to the microcomputer, compiled, and the program was run. In all cases it performed as it had before. A text file transferred to Unix will not have the same length as it did when it was on the microcomputer. This is because the Kermit that runs on Unix takes an arriving carriage return character and line feed character and converts them to just a line feed character. The reverse occurs when the Unix Kermit sends a file unless the command kermit si is used as discussed above under the receive command. The net effect of this is to make a text file transferred to Unix have fewer characters by just the number of lines in the file.

It has been found that approximately 16 packets per minute are sent or received by the Edison-Kermit system. Table 5.2 shows the approximate average number of file characters sent in a data packet.

Sending a Text File:	83 char/data packet
Receiving a Text File:	83 char/data packet
Sending a Source File:	54 char/data packet
Receiving a Source File:	51 char/data packet

Table 5.2: Actual File Characters Per Data Packet.

From this information it is possible to calculate approximately how long it will take to transfer a file to Unix or back to the microcomputer. As an example, the time it takes to send the Edison-Kermit system text to Unix is calculated. The file is approximately 85 pages long, and a page is 512 characters. Therefore the number of packets required will be

```
number of = (85 page)*(512 char/page)/(83 char/packet)
 packets

          = 524 packets.
```

The amount of time this will take is

time = 524 packet/16 packet/min

= 33 minutes.

Table 5.3 compares bit transfer rates and bandwidth usage of MS-DOS Kermit and the Edison-Kermit system for a text file transfer. The actual data transfer rate is the number of file character bits sent each second. The effective data transfer rate is the total number of bits transferred each second, which includes file characters and packet control characters. The effective data transfer rate for MS-DOS Kermit was estimated because it was not certain how many actual characters were sent in transferring a file. The estimate was obtained by proportionality with the actual and effective data transfer rates for Edison-Kermit. To obtain the bandwidth usage, the data transfer rate is divided by the maximum data transfer rate, which in this case is 1200 bits per second (bps).

System	Actual Data Transfer Rate	Effective Data Transfer Rate	Actual Bandwidth Usage	Effective Bandwidth Usage
Edison-Kermit	165 bps	195 bps	.14	.16
MS-DOS Kermit	420 bps estimated	500 bps	.35 estimated	.42

Table 5.3: Performance of Two Kermit Systems.

5.4 Aspects of the Edison-Kermit System.

There are several differences between this implementation of Kermit and the standard Kermit protocol. There are four main differences: carriage return-line feed conversion, control character prefixing if bit seven is on, initialization parameters, and text/binary file send and receive differences. These topics are discussed in the next few paragraphs.

The standard Kermit protocol upon receiving a text file will take a carriage return and line feed and write only the line feed to the file. This procedure is reversed when a text file is sent. The Edison-Kermit system sends and receives files without making changes to the carriage return and line feed characters.

In standard Kermit bit seven is ignored for a text file and used for a binary file [Cru 1984A][Cru 1984B]. The Edison-Kermit system does not ignore bit seven at anytime. Standard Kermit treats bits 0 through 6 the same whether bit seven is on or not. This has the effect of prefixing and adding 64 to characters 128 to 159, and prefixing and subtracting 64 from character 255 just as the characters 0 to 31 and 127 are prefixed and 64 added or subtracted. Edison-Kermit treats characters 0 to 127 the same as standard Kermit, but it sends and receives characters 128 to 255 with no special processing. The implementer realized characters 128 to 255 will have no special effect on either Unix or the PC so there was no reason to have the overhead of a prefixing character. This lack of prefixing explains why more actual file characters are sent in an average Edison-Kermit binary file packet than are sent by the Unix Kermit for the same kind of file. This can be seen in Table 5.2.

Edison-Kermit does nothing with the initialization protocol parameters it receives from the other computer. The values of the parameters that occur in figure 4.2 are set inside the Edison-Kermit system and can only be changed by modifying the code. The values for these parameters are given in Table 5.4 below.

MAXL:	94.
TIME:	20 sec approximately.
NPAD:	0, no padding characters.
PADC:	no padding character.
EOL :	no end of packet character.
QCTL:	#, character used to prefix a control character.
QBIN:	no prefixing character if bit seven is on.
CHKT:	1, a single character checksum.
REPT:	no repeated character prefix.

Table 5.4: Edison-Kermit Protocol Parameters.

Edison-Kermit has one difference in the way it treats binary files compared with text files. An Edison-Kermit text file consists of characters and therefore the program only sends and receives single and separate bytes. A binary file contains integers, and an integer in Edison is two bytes long. This means that when a binary file is sent or received the bytes are treated in pairs. If the high byte of the pair has its most significant bit on, the integer will be negative.

Chapter 6. Edison-Kermit Implementation.

6.1 Introduction.

This chapter is divided into two sections. The first section discusses the entry procedures that were added to the Edison-Kermit kernel. The second section deals with the changes that were made to the Edison-Kermit operating system.

6.2 Procedures Added to the Edison Kernel.

Five new procedures were added to the kernel which are entry points to the kernel from the operating system. In the Edison-Kermit operating system they are called init_port, send_port, port_rec, test_key, and test_port. The following paragraphs will give a brief description of what each of these procedures does.

Init_port. This procedure receives an integer value through its pass by value parameter. This procedure uses the parameter value to set the serial port's baud rate, number of stop bits, parity, and byte length [Zen 1984].

Send_port. This procedure receives a character in its pass by value parameter. It places this character in the serial port's send register.

Port_rec. This procedure obtains an integer from the serial port. In the Edison-Kermit system an integer is two bytes long. The high byte contains information on the status of the port, and the low byte is the ASCII value of the character at the port. This integer is placed in the procedure's pass by reference parameter.

Test_key. This procedure tests the keyboard to see if a key has been pressed. If a key has been pressed, then it returns a 1, otherwise it returns a 0.

Test_port. This procedure tests the serial port and returns the port status in its pass by reference parameter. The port status is an integer and the high byte of this integer contains information about the port. The meaning of the integer value is explained in reference [Zen 1984] of the bibliography.

6.3 The Edison-Kermit Operating System Implementation.

In this section an overview will be given of the additions and changes that were made to the Edison operating system and a top down view of the Edison-Kermit system.

It was necessary to remove some system commands from the Edison operating system so that there would be sufficient room for the new file transfer commands. Besides the commands, some procedures called by the commands were also removed.

Four new modules were added to the operating system and substantial changes were made to the Standard Commands module. The four new modules are Frames, Kermit Utilities, Send and Check Frames, and Receive and Check Frames. These modules are discussed in a general way in the next few paragraphs.

The Frames module has four procedures to manipulate the packet sequence number. The remainder of the procedures create the eight different kinds of packets used by the Edison-Kermit system file transfer protocol.

The Kermit Utilities module has procedures which are used by more than one of the standard commands: connect, send, and receive. These procedures are of a general sort and do such things as send a character to the serial port, receive a character from the serial port, clear the monitor's screen, test the serial port's status, send a packet to the serial port, and receive a packet from the serial port.

The Send and Check Frames module contains procedures that are used by the send procedure in the Standard Commands module. The procedures generally retrieve information from the disk, send packets, wait for the acknowledgment packet that comes back from the remote computer, and check the acknowledgment packet for the correct sequence number and checksum.

The Receive and Check Frames module contains procedures that are used by the receive procedure in the Standard Commands module. The procedures generally receive packets from the remote computer, check the packets for correctness, send acknowledgment packets, and write out the received information to the disk.

The Standard Commands module contains procedures which are called from the main program of the operating system. There is a procedure in this module for all nine commands of the Edison-Kermit system.

In what follows, an overview will be given of the general workings of each of the four new commands.

Initportl. This procedure is the simplest of the four. It queries a user to determine the proper configuration of the serial port. This information is an integer and the value is passed to the init_port kernel entry procedure.

Connect. This procedure is very simple. It is a WHILE loop that continues looping until an ASCII character 17, control q, is read from the keyboard. The loop code tests if the keyboard is active, and if it has been, then the keyboard character is sent to the serial port. Also inside the loop, a character is taken from the port and if a character has arrived, it is displayed on the monitor screen.

Send. The send procedure sends a series of packets and after each packet it waits for a reply from the receiver. If a positive acknowledgment (ack) comes back from the receiver, it tests the checksum and sequence number of the ack packet. If they are correct, it sends the next packet. If they are not correct, it retransmits the last packet. If no response comes back from the receiver after 5 seconds, or a negative acknowledgment is received, then the last packet is sent again. A packet can be sent 5 times before the procedure gives up. This procedure continues through the sequence of packets: send initialization, file header, multiple data packets, end of file packet, and end of transmission packet.

Receive. The receive procedure takes each incoming packet and checks the sequence number, packet type, and checksum value. If the packet's checksum, sequence number, and packet type are correct, it sends a positive acknowlegment, otherwise it sends a negative acknowlegement. The order of the packet types must be send initialization packet, file header packet, data packets, end of file packet, and end of transmission packet. The receive will wait approximately twenty seconds for a response, if no response is received, then it gives up.

Chapter 7. Conclusions and Future Work.

Overall the Edison-Kermit system works well. Of course it is impossible to test the system completely, and perhaps there are problems with the system which have not been discovered yet. If a problem does arise, it should be relatively minor and easily fixed.

One problem for which there is no easy solution is the slowness of a file transfer. This is apparently limited by the execution speed of the Edison code.

A nice improvement to the Edison-Kermit system would be to allow the vi editor to be used while the Edison-Kermit system is in operation. This would permit a file to be transferred to Unix, changes made to the file, and the changed file to be transferred back to the Edison-Kermit system without moving from the terminal or changing the microcomputer to MS-DOS and Perfect Link [Col 1983].

The present Edison System only runs on microcomputers with two floppy disk drives. Its performance would be increased if the system could be loaded onto a partition of a Winchester disk. This would increase the speed of loading the operating system and decrease the access time for reading and writing files from and to the disk.

Assuming the Winchester disk is used for file storage, the number of files allowed will be greatly increased. For a large file catalog, a directory system would be very helpful. Therefore it is suggested that if the Edison System is placed on the Winchester disk, that the file catalog system be expanded to allow directories.

The usefulness of the Edison language could be increased if Edison programs could be run on the KSU Computer Science Department's minicomputers. Another approach to this is to transfer the entire Edison System to Unix and have the Edison operating system run on top of Unix.

One way to accomplish this task would be to change the code generation pass, pass 4, of the Edison compiler so that it generates machine code for the target machine. In addition, it will be necessary to change the kernel from Alva to the assembly language used by the target machine. Another way to accomplish the kernel change is to leave it as is and write a cross assembler which will generate target machine assembly code from Alva assembly code.

Appendix A. User's Manual

1.1 Introduction.

This appendix is a user's manual for operating the Edison-Kermit file transfer system. In the text all user typed commands will be placed on a separate line and computer responses will be placed in double quotes ,i.e., ". The abbreviation <ctl> preceding a character means to hold the control key down and then strike the key for the character. The abbreviation <rtn> stands for the return key. For the expression <file name> you should type a specific file name. This appendix is divided into the following sections: Selecting a Proper Microcomputer, Connecting to the Unix System, Sending a Text File, Receiving a Text File, Sending a Binary File, Receiving a Binary File, and Sample Sessions.

1.2 Selecting a Proper Microcomputer

The Edison-Kermit system will work on an IBM PC microcomputer, or a microcomputer that is compatible with the PC. Currently, in the Kansas State University Computer Science Department, there are Zenith 150 microcomputers and Columbia Data Products PC's that support the Edison-Kermit system. This system will work only with those microcomputers that have two floppy disk drives. The microcomputer selected must have a connection to the Unix system through serial port one, which is sometimes called COM1. If the Unix system has a fixed baud rate line it must be 1200 baud or slower. At higher baud rates the Edison-Kermit system will miss some of the characters that are sent and received.

1.3 Connecting to the Unix System

The Edison-Kermit system disk should be placed in the left or top drive. In MS DOS this is referred to as the A drive, and in the Edison-Kermit system it is drive 0. Another Edison-Kermit formated disk must be placed in the other drive. The computer should be switched on, and the operating system will then be automatically booted.

If you make a mistake in typing a command to Edison-Kermit, then use the backspace key to backup to where the error occurred. Type the command correctly, and then use the delete key to remove the extra characters at the end of the command.

The operating system should write out a header, and then the "Command =" prompt will be displayed. If the computer has just been switched on, the port must be initialized. To initialize the port you should type the command

```
        initport <rtn>
```

and Edison-Kermit will respond with

```
        "Type t for default configuration."
```

If you type

```
        t <rtn>
```

you will initialize the port with this standard configuration. The standard configuration is 1200 baud, 1 stop bit, no parity, and an 8 bit byte. If you type any other letter, Edison-Kermit will come back with

"Configuration value ="

Use Table A1.1 to decide what value to use if you need a different configuration.

Baud rate		
	9600	224
	4800	192
	1200	128
	600	96
	300	64
	150	32
	110	0
parity		
	no parity	0
	odd parity	8
	even parity	24
number of stop bits		
	1	0
	2	4
byte length		
	7 bits	2
	8 bits	3

Table A1.1: Configuration Values.

Add all the values for your choices under each of the four categories. Type this sum for the configuration value. A decision was made not to have the system do this calculation. The reasons for that decision are explained in chapter 5.

Again the Edison operating system will give the prompt

"Command =". To use the microcomputer as a terminal you should type the command

```
        connect <rtn>.
```

Unix should respond with the login and password prompts. Login to the Unix system in the usual way. Unix will request a terminal type. A good terminal emulation type has not been found, so go with the default or vt52. This has no great consequences unless you decide to use the vi editor. Vi is pretty well unusable because of this problem.

After that Unix should give you a prompt, and you should respond with

```
        stty vt05 <rtn>.
```

This sets the terminal so that the first character on each line is not lost.

1.4 Sending a Text file.

Login to Unix as described above and return to the microcomputer by typing

```
        <ctl> q.
```

The file you wish to send must be on the system disk in drive 0 or the disk in drive 1. To see what files are on each disk, type the command

```
        list <rtn>,
```

and Edison-Kermit will respond with "Drive no =". Type

```
        0 or 1 <rtn>
```

to this prompt. If the file you wish to send is not on either disk, insert the proper disk and reboot the system by holding down the keys

```
        <ctl> <shift> and <del>.
```

Again list the files on the disks. Assuming you now have the file you want to send on one of the disks, reconnect to Unix by typing

```
        connect <rtn>,
```

and to the Unix prompt you should type

```
        kermit r <rtn>.
```

You should do the next part relatively quickly because Kermit on Unix can timeout if too much time is taken. Type

```
        <ctl> q
```

to return to the microcomputer. After the Edison-Kermit prompt type

```
        send <rtn>
```

and the system will print "File to be sent =". You should then type

```
        <file name> <rtn>
```

If the file name that you use is in the file catalogs for both disks, the system will use the file on floppy disk 0.

Finally the system will ask "text or binary file = ". You should type

```
        text <rtn>.
```

Sit back and watch the diagnostics.

The system will eventually give the message "Send completed ok", "Send failed", or an error message.

If the send failed then go back to Unix and start by typing

```
        kermit r <rtn>
```

and repeat the steps following that command in the text above.

1.5 Receiving a Text File.

Login to Unix as described above in the section entitled "Connecting to the Unix System". Find the file on Unix that you wish to transfer back to the microcomputer. Go back to the microcomputer by typing

```
        <ctl> q.
```

To the "Command =" prompt type

```
        list <rtn>
```

and Edison-Kermit will respond with "Drive no =". You should type the drive number of the floppy disk where the file is to be sent either

```
        0 or 1 <rtn>.
```

Check to make certain that an Edison-Kermit file of the same name as the one to be sent does not already exist on the drive. If one does then you may rename the file. To do this consult chapter 3 of this report, or you may change the Unix file name by going back to Unix and typing

```
        mv <old file name> <new file name> <rtn>.
```

Once any name conflict is resolved, return to Unix and type

```
        kermit si <filename> <rtn>.
```

Kermit si is used because it prevents the carriage-return/line-feed character conversion that results when kermit s is used. A complete explanation of this can be found in Chapter 5 of this report. The next several commands should be done relatively quickly otherwise Kermit on Unix will time out. You should type

```
        <ctl> q.
```

To the Edison-Kermit prompt you should type

```
        receive <rtn>
```

and the system will ask "text or binary file = " and you should respond with

```
        text <rtn>.
```

Finally the system will ask "Receiving file drive = ". You should respond with

```
        0 or 1 <rtn>
```

depending on which disk you want to receive the file. Sit back and watch the diagnostics and the system will eventually give a message "Receive completed ok", "Receive failed", or an error message.

If the receive failed, connect to Unix again with

```
        connect <rtn>
```

and type

```
        kermit si <file name> <rtn>
```

and repeat the steps following that command in the text above.

1.6 Sending a Binary File.

Sending a binary file is identical with sending a text file except for two command changes. The command kermit r <rtn> should be changed to

```
        kermit ri <rtn>
```

and to the Edison-Kermit prompt "text or binary file = " you should type

```
        binary.
```

Appendix B discusses an error that was found in the Unix Kermit. Until the error in the Unix Kermit is fixed, it is necessary to make a local copy of Kermit and change a line in the Kermit source code. Consult appendix B to see how this is done.

1.7 Receiving a Binary File.

Receiving a binary file is identical with receiving a text file except for one command change. When the Edison-Kermit prompt "text or binary file = " is given you should type

binary.

1.8 Sample sessions.

For the following sessions all Unix and Edison-Kermit responses and the explanations that go with them will be underlined . For these examples it has been assumed that: the microcomputer has just been turned on so it is necessary to initialize the port, the user has not previously logged onto Unix, a text file is to be sent, and the name of the file to be sent or received is systxt. Under the section Receiving a Text File, the Unix command kermit si systxt is given as opposed to the kermit s systxt. The reason for this choice is explained in Chapter 5 of this report. In each of the next two subsections there are starred lines. Those are lines that you must change if a binary file is to be sent instead of a text file. This is explained in section 1.8.3 of this chapter.

1.8.1 Sending a Text File.

RESPONSES EXPLANATION

<u>Command</u> =	<u>Edison-Kermit prompt</u>
initport <rtn>	request to initialize port config. values
<<u>msg about the</u> <<u>default config</u>.>	<u>Edison-Kermit response</u>
t <rtn>	accept default init-ialization of port
<u>Command</u> =	<u>Edison-Kermit prompt</u>
connect <rtn>	request to connect to Unix
<u>login</u>	<u>Unix login prompt</u>
terry <rtn>	type login name
<u>password</u>	<u>Unix password prompt</u>
billbo <rtn>	type password
<<u>several messages</u>>	<u>Unix login messages</u>
<u>term type</u> =	<u>Unix asks for term. type</u>
vt52 <rtn>	user vt52 emulation
%	<u>Unix prompt</u> (<u>C-shell</u>)
stty vt05 <rtn>	set Unix so it prints the first character
%	<u>Unix prompt</u>
* kermit r <rtn>	start Kermit running on Unix, do not type the asterisk.
<ctl> q	go back to micro
<u>Command</u> =	<u>Edison-Kermit prompt</u>
send <rtn>	request to send file
<u>File to be sent</u> =	<u>Edison Kermit prompt</u>
systemtext <rtn>	name of file to send

text or binary file =	_what kind of file_
* text <rtn>	text file to be sent, do not type the asterisk
<_Edison-Kermit diagnostics_>	_messages on file transfer_
file transfer ok	_system reports transfer ok_
Command =	_Edison-Kermit prompt_
connect <rtn>	hook up to Unix
%	_Unix prompt_
ll	make certain the file arrived okay

1.8.2 _Receiving a Text File_.

RESPONSES	_EXPLANATION_
Command =	_Edison-Kermit prompt_
initport <rtn>	request to init port values
<_msg about the default config_.>	_Edison-Kermit response_
t <rtn>	accept default init. of port
Command =	_Edison-Kermit prompt_
connect <rtn>	request to connect to Unix
login	_Unix login prompt_
terry <rtn>	type login name

password	Unix password prompt
billbo <rtn>	type password
<several messages>	Unix login messages
term type =	Unix asks for term. type
vt52 <rtn>	user vt52 emulation
%	Unix prompt (C-shell)
stty vt05 <rtn>	set Unix so it prints the first character
%	Unix prompt
kermit si systxt <rtn>	start Kermit running on Unix,
<ctl> q	go back to micro
Command =	Edison-Kermit prompt
receive <rtn>	request to receive file
text or binary file =	what kind of file
* text <rtn>	text file, do not type the asterisk
Receiving file drive =	Edison Kermit prompt
1 <rtn>	place file on drive 1
<Edison-Kermit diagnostics>	messages on file transfer
receive completed ok	system reports transfer ok

Command =	Edison-Kermit prompt
list <rtn>	look at file directory
Which drive =	Edison-Kermit prompt
1 <rtn>	look at directory drive 1

1.8.3 Sending and Receiving Binary Files.

You need make at most two changes to the sample sessions to send or receive binary files. In the above sample sessions a star has been placed in front of each line where a change must be made. The command

```
        text
```

must be changed to

```
        binary,
```

and the command

```
        kermit r
```

must be changed to

```
        kermit ri
```

Appendix B. Hints for Edison Programmers.

2.1 Introduction.

This appendix will try to help future Edison programmers by discussing information which was learned the hard way while completing this project. This appendix is broken into three parts: information that will be helpful while using the standard Edison System, information that can be used when making changes to the Edison operating system and kernel, and information that can help a programmer trying to use the serial port on the microcomputer and Kermit on the micro and Unix.

2.2 Helpful Hints On Using Standard Edison.

The operating system commands: copy, newsystem, and newkernel and all the auxiliary programs do not ask the drive number where a file is to be found. The system begins searching for a file on disk 0. If it finds a file of that name on disk 0 then it does not check any further. This of course can lead to problems if the file you want to use is on disk 1 and a file with the same name is on disk 0. It is recommended that name ambiguity be checked for before you begin any of the operations listed above.

Chapter 3 of this report discusses some things to be aware of while using the Edison editor. Those suggestions are repeated here. After editing a file the editor will ask if you want the changes saved. You must respond with the word yes if the changes are to be saved. Anything other than the word yes and the changed file will not be written to the disk.

Before using the editor, it is important to check to see how many free pages are left on the disk where the file is to be saved. For a file to be saved, there must be at least as many free pages on the disk as the size of the file to be saved.

A final precaution is to check for a name ambiguity. If you try to save a file on a disk where that file name already exists, then the file will not be saved. For example consider the case where you edit a file that originally was on disk 0. If there is not enough room to save the file on that disk, then you must save it on disk 1. However, if that file name already exists on disk 1, the edited file will not be saved. The easiest way to prevent this problem is to check the file catalog before editing, and make certain there will be no name ambiguity.

Two common compile errors are misplacement of a semicolon and name ambiguity. The semicolon usage in Edison is different from Pascal. Make certain you place a semicolon between statements, but never before the reserved words proc, var, begin, end, array, const, record, else, or module. Another possible error to watch out for is name ambiguity, which can occur if two identical names are exported from different modules.

The readsector procedure in the prefix can be very useful for reading a disk sector. If you should use this procedure to read the file catalog, remember that the parameter list of readsector contains a sector number, but what is read in the file catalog is a page number. To convert a page number to a sector number use the formula

sectornumber = (pagenumber - 1) * 2.

2.3 Helpful Hints When Changing the Edison Operating System and Kernel.

If you need to make changes to the Edison operating system or kernel, this section has information which was discovered while doing this project.

The compiler can fail to work correctly because a limit is exceeded in the compiler. The compiler failures are listed on pages 139-140 of reference [Bri 1982] in the bibliography. To fix this sort of error, identify the reason for the failure, increase the size of the constant which caused the failure, and recompile the compiler textfile where the change was made.

The BIOS interrupt routines are called in the kernel. The Programmer's Utility Guide [Zen 1984] has a very clear description of what each of the interrupts does and should answer most questions you might have about the BIOS routines. The only problem encountered in making changes to the kernel was discovering that the kernel maxparameter constant must be increased by one every time a procedure is added to the list of other kernel entry procedures.

If procedures are added to the prefix of the operating system, then all programs that are used with that operating system must have their prefix changed to agree with the new prefix.

The operating system must fit on pages 7 through 24 of the system disk. If the operating system becomes too large, it may become necessary to increase the number of pages given to the operating system. It may be tempting to increase the upper page limit to more than 24, but then this will write over the disk catalog, and this will cause a lot of problems.

The easiest solution is to eliminate unnecessary operating system procedures until the system fits on the normal pages. However, if this will not work, the next best alternative is to change the kernel so that it expects to find the operating system at the top end of the page numbers. The operating system loader is found at the end of kerneltext1 and it is only necessary to change the progsector constant from 7 to the new value. The newsystem procedure in the operating system must also be changed so that the operating system is placed at the proper location on the floppy disk. The systemaddr and systemlimit constants in the newsystem procedure must be changed to the new starting page number and the new length of the operating system.

One final problem that occurred was a name overflow. This is caused by overflowing the store. When a program is run, the variables and constants are placed in the store. Every new block that is entered has its local variables added to the store. If there are too many global variables, then when a new block with too many local variables is entered, there will not be sufficient room in the store for the local variables. The only easy cure for this problem is to try to eliminate as many variables as possible, particularly global variables.

2.4 Helpful Hints on the Serial Port, Kermit, and Unix.

In the process of writing the Edison-Kermit system some information was gained that might be helpful to persons interested in this part of the project.

Once the baud rate on the serial port was set, it was thought that sending characters to the port and receiving characters from the port would always be done at the proper rate. That is not true and it is up to the programmer to make certain that characters are not overrun when they are sent, nor extraneous characters taken from the port.

The receive procedure returns an integer which contains character and port status information. The port status information reports when no character has arrived at the port within one second. When the receive procedure was first written, only character information was returned.

Written this way the procedure appeared to receive an ASCII character 96 when no character had arrived at the port.

For sending characters it is necessary to have a separate kernel procedure which tests the port and makes certain the last character has been completely sent before a new character is placed in the send register. One difficulty that was encountered is that the port test procedure can not be used without a time delay between calls. If the time delay is not long enough, then the test_port procedure does not work correctly. The solution was to use a delay loop which is done before a call to the test port procedure. If the delay time is not long enough, then Edison-Kermit will stall out, since the test_port procedure will always report that the send register is not empty.

One very big help in working on the port communications was a cable that was used to connect between the serial ports of two microcomputers. Once the serial port procedures were written, they were tested by loading the modified operating systems on the two microcomputers and testing to see if characters could be sent between the two computers.

A cable was also used when testing the Edison-Kermit procedures. This cable was connected to Unix and three microcomputers. One of the micros served as a terminal and the other two were used to monitor what was sent by the terminal micro and by Unix. In this way the actual Kermit packets sent by the terminal and by Unix could be observed.

Once logged onto Unix through the serial port the command, stty vt05, must be given. This command has Unix delay sending the next character after a carriage return. In Edison there is a delay after a carriage return, and because of this, the first character of a line will be lost at 1200 baud unless the stty vt05 command is given.

When the microcomputer is first connected to Unix, a carriage return, i.e., CHAR(13) must be sent before the login prompt is displayed. You must use a CHAR(13) rather than a CHAR(10) because Unix uses this first character to determine what to send for the new line character. If CHAR(10) is sent, only a line feed and not a carriage return will come back from Unix.

The Unix Kermit has a coding error which was discovered when trying to have Unix receive a binary file. The Unix Kermit requires an i to be added to the command line when a binary file is to be sent or received. This i should force the Kermit Protocol to use bit seven of each byte, but in the Kermit program bit seven is automatically made zero without consideration of whether an i appeared on the command line. A local copy of Kermit was changed by adding an if statement to the Kermit program so that bit seven is zeroed only if an i does not appear on the command line. To change a local copy for your own use, copy or mail the Kermit source code from the Interdata 3220 /usrb/wb/kermit/kermit2.c to your own account. Change the code in procedure xread from

```
buf[i] &= 0x7f;  to    if(!image) buf[i] &= 0x7f;
```

In Unix you type cc kermit2.c which will compile the code and produce object code in a.out. If is is necessary to have Unix receive a binary file, you must use this local copy of Kermit.

The problem that took the longest time to discover occurred when Edison-Kermit was to send a relatively long file. The transfer usually failed when the first data frame was sent. The Unix Kermit was timing out because the Edison-Kermit system was slow in sending that first data frame. The fix for this problem was to tell the Unix Kermit to wait longer before timing out. This is done by sending the needed timeout period in the send initialization packet. See chapter 4 of this report for details of how this is done.

Appendix C: Sample Kermit Session.

The following is an example of a Kermit session.

Packet Sent	Explanation
^A) SH(@-#^	send init
^A) YH(@-#%	ack for send init
^A+!FMOON.DOC2	file header
^A#!Y?	ack for file header
^AE"D No celestial has required J	first data frame
^A#"Y@	ack for data frame
:	:
:	:
^AE#Das much labor for th%%%tudy of its#	bad data frame
^A##N8	nak for bad frame
:	:
:	:
^A##ZB	eof frame

^A##YA	ack for eof frame
^A#$B+	eot frame
^A#$YB	ack for eot frame

The ^A at the beginning of each packet stands for SOH, start of header. Chapter 4 of this report gives the fields for each packet, but in short the fields are: SOH, Len, Seq, Type, Data, Checksum. The following table will explain the values of each of the fields for the first packet above. The symbols sp stand for the space character.

Packet	Field Name	Subfield Name	Meaning
^A	SOH		start of header
(	Length		41 - 32 = 9 chars following length field
sp	Seq Num		32 - 32 = 0 first packet
S	Packet Type		send init. pkt.
H	Data	max pkt. len.	72 - 32 = 40
(	Data	timeout period	40 - 32 = 8 sec.
sp	Data	num. padding characters	32 - 32 = 0
@	Data	padding char.	@

-	Data	end of line character	45 - 32 = 13 carriage rtn char.
#	Data	control quote character	prefix for a control char.
^	Checksum		checksum is 94 ASCII character ^

The checksum is obtained in the following manner.

characters	)	sp	S	H	(	sp	@	-	#
ASCII values	41	32	83	72	40	32	64	45	35

Notice the SOH and checksum fields were not included. The total of the ASCII values is 444. Moding this with 256 will keep only the low byte. The low byte is 188 base ten, and this converted to base two is 10111100. Adding bits 7 and 6 to bits 5 through 0, the result 111110 is obtained. This number in base ten is 62, and adding 32 to this gives 94.

The next frame is just the acknowledgment for the "Send Initialization" packet. The next frame sends the file name, which in this case is MOON.DOC. A series of data frames are sent, until a frame is sent which is corrupted in transmission. The receiver sends a nak for this packet, and the sender retransmits the packet. When all of the data have been sent, an end of file packet is transmitted. This packet is acknowledged and an end of transmission is sent and acknowledged by the receiver.

APPENDIX D

This appendix contains the modified kernel code used with the Edison-Kermit system.

```
$ "Edison kernel for the IBM Personal Computer
$       and the Compaq Portable Computer
$                 Part 1 of 2
$              18 August 1982
$           (Revised 20 May 1983)
$     Copyright (c) 1982 Per Brinch Hansen"
$     Changes made 1985 by Terry Scott indicated by *'s

base 1280
do start:                      $ proc kernel
  jump(disks)
                               $ const overflow = false;
const false = 0                $   baseaddr = 640;
const true = 1                 $   maxaddr = 9999
const maxaddr = #177774
                               $ array store
reg ax(0)                      $   [0:maxaddr] (int)
reg bx(3)
reg cx(1)                      $ var st: store;
reg dx(2)                      $   b, s, t, p: int
reg sp(4)
reg  b(5)
reg  s(6)                      $ pre proc diskproblem
reg  p(7)
word t
                               $ module "disks"
const extendsign = #231
const sectorlength = 512       $ * const sectorlength = 256
const sector = 256             $ * array sector
                               $     [1:sectorlength] (int)

const drive = 0                $ * const drive = 0;
const input = 513              $     input = 513;
const output = 769             $     output = 769

word loaded                    $ var loaded: bool;
word errors                    $   errors: int

const diskreset = 0            $ proc diskerror
do diskerror:                  $ begin
  move(diskreset, ax)          $   "BIOS_diskreset;"
  interrupt(19)                $   if loaded do
  compare(true, st[loaded])    $     errors := errors + 1;
  ifnotequal(diskerror2)       $     if errors = 3 do
  increment(st[errors])        $       diskproblem;
  compare(3, st[errors])      $       errors := 0
  ifnotequal(diskerror2)      $     end
  call(diskproblem)           $   end
  move(0, st[errors])         $ end
do diskerror2:
  return

word headno                   $ * proc diskio(
word trackno                  $     operation,
word sectorno                 $     driveno,
do diskio:                    $     sectorno,
  move(sp, bx)                $     address: int)
  move(st[bx+4], ax)          $ var headno, trackno: int;
  instr(extendsign)           $   again: bool
  move(320, cx)               $ begin
  divide(cx)                  $   "0 <= sectorno <= 639"
  move(ax, st[headno])        $   headno :=
  move(st[bx+4], ax)          $     sectorno div 320;
  instr(extendsign)           $   trackno :=
  move(80, cx)                $     sectorno mod 80 div 2;
  divide(cx)                  $   sectorno :=
  halve(dx)                   $     sectorno div 80 mod 4
  move(dx, st[trackno])       $       + sectorno mod 2 * 4
  and(3, ax)                  $         + 1;
  move(st[bx+4], dx)          $   errors := 0;
  and(1, dx)                  $   again := true;
  double(dx)                  $   while again do
  double(dx)                  $     "BIOS_diskio(
  add(dx, ax)                 $        operation,
  increment(ax)               $        driveno,
  move(ax, st[sectorno])      $        headno,
  move(0, st[errors])         $        trackno,
do diskio2:                   $        sectorno,
  move(sp, bx)                $        address,
  move(st[headno], dx)        $        again);"
  move(8, cx)                 $     if again do
  shiftleft(dx)               $       diskerror
  add(st[bx+6], dx)           $     end
  move(st[trackno], ax)       $   end
  move(8, cx)                 $ end
  shiftleft(ax)
  add(st[sectorno], ax)
  move(ax, cx)
  move(st[bx+8], ax)
  move(st[bx+2], bx)
  interrupt(19)
  ifnotcarry(diskio3)
  call(diskerror)
  jump(diskio2)
do diskio3:
  return(8)

const formatl = 1280          $ * proc formattrack(
const sectortype = 512        $     driveno, trackno: int)
const sectors = 8               $ const sectortype = 2;
array table [15]                $   sectors = 8
word endtable                   $ "packed" record item(
word headno2                    $    track_no,
word trackno2                   $    head_no,

do formattrack:                 $    sector_no,
  move(sp, bx)                  $    sector_type: int
  move(st[bx+2], ax)            $ array list [1:8] (item)
  instr(extendsign)             $ var table: list;
  move(40, cx)                  $   headno, sectorno: int;
  divide(cx)                    $   again: bool
  move(ax, st[headno2])         $ begin
  move(dx, st[trackno2])        $   "0 <= trackno <= 79"
  move(8, cx)                   $   headno :=
  shiftleft(ax)                 $     trackno div 40;
  add(dx, ax)                   $   trackno :=
  move(sectortype, cx)          $     trackno mod 40;
  move(table, bx)               $   sectorno := 1;
do formattrack2:                $   while sectorno <= 8 do
  increment(cx)                 $     table[sectorno] :=
  move(ax, st[bx])              $       item(trackno,
  move(cx, st[bx+2])            $          headno,
  add(4, bx)                    $          sectorno,
  compare(endtable, bx)         $          sectortype);
  ifnothigher(formattrack2)     $     sectorno :=
  move(0, st[errors])           $       sectorno + 1
do formattrack3:                $   end;
  move(sp, bx)                  $   errors := 0;
  move(st[headno2], dx)         $   again := true;
  move(8, cx)                   $   while again do
  shiftleft(dx)                 $     "BIOS_format(driveno,
  add(st[bx+4], dx)             $        trackno, headno,
  move(st[trackno2], ax)        $        table, sectors,
  move(8, cx)                   $        again);"
  shiftleft(ax)                 $     if again do
  move(ax, cx)                  $       diskerror
  move(table, bx)               $     end
  move(format1, ax)             $   end
  add(sectors, ax)              $ end
  interrupt(19)
  ifnotcarry(formattrack4)
  call(diskerror)
  jump(formattrack3)
do formattrack4:
  return(4)

const loadaddr = 31744          $ * const minaddr = 999
word addr                       $ var addr, sectorno: int
word sectorno2                  $ begin
do disks:                       $     "Input   this   module
  move(0, ax)                   $   from  the autoload sector
  move_ss(ax)                   $   (sector 0 on disk 0)  and
  move_ds(ax)                   $   place  it   at  the  load
  move_es(ax)                  $   address (31744);
  move(maxaddr, sp)            $     Set  the  four  segment
  move(loadaddr, s)            $   registers to zero;
  move(start, p)               $     Place the kernel  stack
  move(sector, cx)             $   temporarily at maxaddr;
do disks2:                     $     Move    the    autoload
  move(st[s], ax)              $   sector   to   the    base
  move(ax, st[p])              $   address  (1280)   -   the
  add(2, s)                    $   first address  after  the
  add(2, p)                    $   BIOS data area;
  loop(disks2)                 $     Input  the  rest of the
  jump_cs(0, disks3)           $   kernel;"
do disks3:                     $   loaded := false;
  move(false, st[loaded])      $   sectorno := 0;
  move(0, st[sectorno2])       $   addr := baseaddr;
  move(start, st[addr])        $   while addr < minaddr do
do disks4:                     $     sectorno :=
  compare(minaddr, st[addr])   $       sectorno + 1;
  ifhigher(disks5)             $     addr :=
  increment(st[sectorno2])     $       addr + sectorlength;
  add(sectorlength, st[addr])  $     diskio(input, drive,
  move(input, ax)              $       sectorno, addr)
  push(ax)                     $   end;
  move(drive, ax)              $   loaded := true
  push(ax)                     $     "Place the kernel stack
  push(st[sectorno2])          $   at its final position"
  push(st[addr])               $ end "disks"
  call(diskio)
  jump(disks4)
do disks5:
  move(true, st[loaded])
  move(stackbottom, sp)
  jump(screen)                 $ "end of autoload sector"

pad 1792                       $ enum opcode (
do opcode:                     $   "standard codes"
  jump(addx)                   $   add4,
  jump(alsox)                  $   also4,
  jump(andx)                   $   and4,
  jump(assign)                 $   assign4,
  jump(blank)                  $   blank4,
  jump(cobeginx)               $   cobegin4,
  jump(constant)               $   constant4,
  jump(construct)              $   construct4,
  jump(difference)             $   difference4,
  jump(dividex)                $   divide4,
  jump(dox)                    $   do4,
  jump(elsex)                  $   else4,
  jump(endcode)                $   endcode4,
  jump(endlib)                 $   endlib4,
  jump(endproc)                $   endproc4,
  jump(endwhen)                $   endwhen4,
  jump(equal)                  $   equal4,
  jump(field)                  $   field4,
     jump(goto)                         $   goto4,
     jump(greater)                      $   greater4,
     jump(inx)                          $   in4,
     jump(index)                        $   index4,
     jump(instance)                     $   instance4,
     jump(intersection)                 $   intersection4,
     jump(less)                         $   less4,
     jump(libproc)                      $   libproc4,
     jump(minusx)                       $   minus4,
     jump(modulo)                       $   modulo4,
     jump(multiplyx)                    $   multiply4,
     jump(newline)                      $   newline4,
     jump(notx)                         $   not4,
     jump(notequal)                     $   notequal4,
     jump(notgreater)                   $   notgreater4,
     jump(notless)                      $   notless4,
     jump(orx)                          $   or4,
     jump(paramarg)                     $   paramarg4,
     jump(paramcall)                    $   paramcall4,
     jump(procarg)                      $   procarg4,
     jump(proccall)                     $   proccall4,
     jump(procedure)                    $   procedure4,
     jump(process)                      $   process4,
     jump(subtractx)                    $   subtract4,
     jump(union)                        $   union4,
     jump(valspace)                     $   valspace4,
     jump(value)                        $   value4,
     jump(variable)                     $   variable4,
     jump(wait)                         $   wait4,
     jump(whenx)                        $   when4,
     jump(addrx)                        $   addr4,
     jump(haltx)                        $   halt4,
     jump(obtainx)                      $   obtain4,
     jump(placex)                       $   place4,
     jump(sensex)                       $   sense4,

                                        $   "extra codes"
     jump(elemassign)                   $   elemassign4,
     jump(elemvalue)                    $   elemvalue4,
     jump(localcase)                    $   localcase4,
     jump(localset)                     $   localset4,
     jump(localvalue)                   $   localvalue4,
     jump(localvar)                     $   localvar4,
     jump(outercall)                    $   outercall4,
     jump(outercase)                    $   outercase4,
     jump(outerparam)                   $   outerparam4,
     jump(outerset)                     $   outerset4,
     jump(outervalue)                   $   outervalue4,
     jump(outervar)                     $   outervar4,
     jump(setconst)                     $   setconst4,
     jump(singleton)                    $   singleton4,
     jump(stringconst)                  $   stringconst4)

align
```

```
array stack [99]
word stackbottom
                               $ const
const bel = 7                  $   bel = char(7);
const lf = 10                  $   lf = char(10);
const cr = 13                  $   cr = char(13);
const nl = lf                  $   nl = lf;
const space = ' '              $   sp = ' ';
const none = 0                 $   none = 0;
const pdpll = false            $   pdpll = false;
const cursorno = 0             $   cursorno = 0;
const eraseno = 1              $   eraseno = 1;
const displayno = 2            $   displayno = 2;
const acceptno = 3             $   acceptno = 3;
const printno = 4              $   printno = 4;
const getno = 5                $   getno = 5;
const putno = 6                $   putno = 6;
const formatno = 7             $   formatno = 7;

                               $ ********************************
const initportno = 8           $ * initportno = 8;              *
const sendportno = 9           $ * sendportno = 9;              *
const recportno = 10           $ * recportno = 10;              *
const testkeyno = 11           $ * testkeyno = 11;              *
const testportno = 12          $ * testportno = 12;             *
const maxparam = 13            $ * maxparam = 13 "procs" 8 to 13 *
const paramlength = 58         $ * paramlength = 19 wrds;38to 58 *
                               $ ********************************
const namelength = 24          $ const namelength = 12
const name = 12                $ array name [l:namelength](char)

                               $ const textlength = 80
                               $ array text [l:textlength](char)

const setlimit = 127           $ const setlimit = 127
const setlength = 16           $ set settypel (int)
const settype = 8              $ const setlength = 8
                               $ array settype2[l:setlength] (int)

const maxproc = 20             $ const maxproc = 20
const maxproc = 20             $ const maxproc = 20
const statelength = 8          $ record procstate(bx,sx,tx,px:int)
const queue = 80
                               $ array queue [l:maxproc](procstate)

array q [queue]                $ var q: queue;
word this                      $   this: int;
word tasks                     $   tasks: int;
word stacktop                  $   stacktop: int;
word progtop                   $   progtop: int;
word blocked                   $   blocked: bool


                               $ module "screen"
```

```
const maxrow = 25
const maxcolumn = 80             $ * const maxrow = 25;
const pageno = 0                 $     maxcolumn = 80

const putl = 512                 $ * proc putcursor(
do putcursor:                    $     row, column: int)
  move(sp, bx)                   $ begin
  move(putl, ax)                 $   "1 <= row <= maxrow,
  move(st[bx+4], dx)             $    1 <= column <=
  decrement(dx)                  $            maxcolumn"
  move(8, cx)                    $   row := row - 1;
  shiftleft(dx)                  $   column := column - 1
  add(st[bx+2], dx)              $   "BIOS_putcursor(
  decrement(dx)                  $      row, column)"
  move(pageno, bx)               $ end
  push(b)
  interrupt(16)
  pop(b)
  return(4)

const getl = 768                 $ * proc getcursor(
do getcursor:                    $     var row, column: int)
  move(getl, ax)                 $ begin
  move(pageno, bx)               $   "BIOS_getcursor(
  push(b)                        $      row, column);"
  interrupt(16)                  $   row := row + 1;
  pop(b)                         $   column := column + 1
  move(dx, ax)                   $   "1 <= row <= maxrow,
  move(8, cx)                    $    1 <= column <=
  shiftright(ax)                 $            maxcolumn"
  increment(ax)                  $ end
  and(#377, dx)
  increment(dx)
  move(sp, cx)
  move(cx, bx)
  move(st[bx+4], bx)
  move(ax, st[bx])
  move(cx, bx)
  move(st[bx+2], bx)
  move(dx, st[bx])
  return(4)

const portl = 0                  $**********************************
do initportl:                    $ * proc initportl(value: int)    *
  move(portl, dx)                $   const portl = 0 "set to COM1" *
  move(sp, bx)                   $   begin                         *
  move(st[bx+2], ax)             $     Bios_configport(portl,value)*
  push(b)                        $   end                           *
  interrupt(20)                  $**********************************
  pop(b)
  return(2)

const send = 256                 $*********************************
do sendportl:                    $ * proc sendportl(value: char)  *
```

```
  move(portl, dx)                  $   begin                        *
  move(sp, bx)                     $     "Bios_sendport(value)      *
  move(send, ax)                   $   end                          *
  add(st[bx+2],ax)                 $********************************
  push(b)
  interrupt(20)
  pop(b)
  return(2)

const testk = 256                  $********************************
do testkeyl:                       $ * proc testkeyl(var value: int)*
  move(testk, ax)                  $   var num: int                 *
  interrupt(22)                    $   begin                        *
  ifnotequal(testkey2)             $     "Bios_testkey(num);"       *
  move(#0, ax)                     $     if num = 0 do              *
  jump (testkey3)                  $       value = 0                *
do testkey2:                       $     else true do               *
  move(#1, ax)                     $       value = 1                *
do testkey3:                       $     end                        *
  move(sp, bx)                     $   end                          *
  move(st[bx+2], bx)               $********************************
  move(ax, st[bx])
  return(2)

const receive = 512                $********************************
do recportl:                       $ * proc recportl(var value: int)*
  move(portl, dx)                  $   begin                        *
  move(receive, ax)                $     "Bios_recport(value)"      *
  interrupt(20)                    $   end                          *
  move(sp, bx)                     $********************************
  move(st[bx+2], bx)
  move(ax, st[bx])
  return(2)

const testp = 768                  $********************************
do testportl:                      $ * proc testportl(var value:int)*
  move(testp, ax)                  $   begin                        *
  interrupt(20)                    $     "Bios_testport(value)"     *
  move(sp, bx)                     $   end                          *
  move(st[bx+2], bx)               $********************************
  move(ax, st[bx])
  return(2)

const writel = 3584                $ * proc write(value: char)
do write:                          $ begin
  move(sp, bx)                     $   "BIOS_writetty(value)"
  move(writel, ax)                 $      skip
  add(st[bx+2], ax)                $ end
  move(pageno, bx)
  push(b)
  interrupt(16)
  pop(b)
  return(2)
```

```
do writechar:                     $ * proc writechar(
  move(sp, bx)                    $     value: char)
  push(st[bx+2])                  $ begin
  compare(lf, st[bx+2])           $   if value = lf do
  ifnotequal(writechar2)          $     write(cr)
  move(cr, ax)                    $   end;
  push(ax)                        $   write(value)
  call(write)                     $ end
do writechar2:
  call(write)
  return(2)

const period = '.'                $ * proc writetext(
do writetext:                     $     value: text)
  move(sp, bx)                    $ const period = '.'
  move(st[bx+2], bx)              $ var i: int; c: char
  move(st[bx], ax)                $ begin
  compare(period, ax)             $   i := 1;
  ifequal(writetext2)             $   c := value[1];
  push(ax)                        $   while c <> period do
  call(writechar)                 $     writechar(c);
  move(sp, bx)                    $     i := i + 1;
  add(2, st[bx+2])                $     c := value[i]
  jump(writetext)                 $   end
do writetext2:                    $ end
  return(2)

word il                           $ * proc writeint(value: int)
do writeint:                      $ array table [1:6] (char)
  move(sp, bx)                    $ var no: table; i: int
  move(0, st[il])                 $ begin "value >= 0"
  move(st[bx+2], ax)              $   if value = 0 do
do writeint2:                     $     i := 1; no[1] := '0'
  increment(st[il])               $   else value > 0 do
  instr(extendsign)               $     i := 0;
  move(10, cx)                    $     while value > 0 do
  divide(cx)                      $       i := i + 1;
  add('0', dx)                    $       no[i] := char(value mod
  push(dx)                        $         10 + int('0'));
  compare(0, ax)                  $       value := value div 10
  ifgreater(writeint2)            $     end
do writeint3:                     $   end;
  call(writechar)                 $   while i > 0 do
  decrement(st[il])               $     writechar(no[i]; i := i - 1
  ifnotequal(writeint3)           $   end
  return(2)                       $ end
word i2                           $ * proc writename(
do writename:                     $     value: name)
  move(name, st[i2])              $ var i: int; c: char
do writename2:                    $ begin
  move(sp, bx)                    $   i := 0;
  move(st[bx+2], bx)              $   while i < namelength do
  move(st[bx], ax)                $     i := i + 1;
  compare(space, ax)              $     c := value[i];
```

```
   ifequal(writename3)          $     if c <> sp do
   push(ax)                     $       writechar(c)
   call(writechar)              $     end
 do writename3:                 $   end
   move(sp, bx)                 $ end
   add(2, st[bx+2])
   decrement(st[i2])
   ifnotequal(writename2)
   return(2)

 const write_ac = 2304          $ * proc erasescreen
 const attribute = 7            $ var row, column,
 word row                       $   positions: int
 word column                    $ begin
 do erasescreen:                $   getcursor(row, column);
   move(row, ax)                $   positions :=
   push(ax)                     $     (maxrow - row + 1)
   move(column, ax)             $       * maxcolumn
   push(ax)                     $         - column + 1
   call(getcursor)              $   "BIOS_write_ac(
   move(maxrow, ax)             $      positions, sp)"
   subtract(st[row], ax)        $ end
   increment(ax)
   move(maxcolumn, cx)
   multiply(cx)
   subtract(st[column], ax)
   increment(ax)
   move(ax, cx)
   move(write_ac, ax)
   add(space, ax)
   move(pageno, bx)
   add(attribute, bx)
   interrupt(16)
   return

 do beep:                       $ * proc beep
   move(bel, ax)                $ begin write(bel) end
   push(ax)
   call(write)
   return

 $ "The screen initialization has been slightly modified to make
 $  the kernel work on both the IBM-PC and the Compaq Portable"

 const SetBW80x25 = 2           $ begin
 do screen:                     $   "BIOS_setmode(
   push(b)                      $      BW80x25)"
   move(SetBW80x25, ax)         $   skip
   interrupt(16)                $ end "screen"
   pop(b)
   jump(keyboard)

                                $ module "keyboard"
```

```
const nul = 0                 $ const nul = char(0);
const del = 127               $   del = char(127)

const readl = 0               $ * proc readx(
do read:                      $     var value: char)
  move(readl, ax)             $ var keyno: int
  interrupt(22)               $ begin
  move(ax, dx)                $   "BIOS_read(keyno,
  and(#377, ax)               $      value);"
  move(8, cx)                 $   if value = nul do
  shiftright(dx)              $     if (59 <= keyno) and
  and(#377, dx)               $       (keyno <= 82) do
  compare(nul, ax)            $         value :=
  ifnotequal(read3)           $           char(keyno - 57)
  compare(59, dx)             $     else keyno = 83 do
  ifless(read2)               $       value := del
  compare(82, dx)             $     end
  ifgreater(read2)            $   else value > del do
  move(dx, ax)                $     value := nul
  subtract(57, ax)            $   end
  jump(read4)                 $ end
do read2:
  compare(83, dx)
  ifnotequal(read4)
  move(del, ax)
  jump(read4)
do read3:
  compare(del, ax)
  ifnotgreater(read4)
  move(nul, ax)
do read4:
  move(sp, bx)
  move(st[bx+2], bx)
  move(ax, st[bx])
  return(2)

const teststatus = 256        $ * proc ready: bool
do ready:                     $ begin
  move(teststatus, ax)        $   "BIOS_teststatus(
  interrupt(22)               $      val ready);"
  move(sp, bx)                $        skip
  move(false, st[bx+2])       $ end
  ifequal(ready2)
  increment(st[bx+2])
do ready2:
  return

text pausetext =              $ * proc pause
  'Push RETURN to continue.'  $ var response: char
word response                 $ begin
do pause:                     $   writetext(text(
  move(pausetext, ax)         $     'Push RETURN to',
  push(ax)                    $      ' continue.'));
  call(writetext)             $   readx(response);
  move(response, ax)           $   writechar(nl)
  push(ax)                     $ end
  call(read)
  move(nl, ax)
  push(ax)
  call(writechar)
  return

do keyboard:                   $ begin skip
  jump(printer)                $ end "keyboard"

const printerno = 0            $ module "printer"

const printl = 0               $ * proc printchar(
do printchar:                  $     value: char)
  move(sp, bx)                 $ begin
  move(printl, ax)             $   "BIOS_print(value)"
  add(st[bx+2], ax)            $      skip
  move(printerno, dx)          $ end
  interrupt(23)
  return(2)

const reset2 = 256             $ begin
do printer:                    $   "BIOS_resetprinter"
  move(reset2, ax)             $      skip
  move(printerno, dx)          $ end
  interrupt(23)
  jump(failure)

word b0                        $ module "program failure"
word s0
word t0
word p0

do loadname:                   $ proc loadname(addr: int;
  push(s)                      $   var value: name)
  move(sp, bx)                 $ var i: int
  move(st[bx+6], s)            $ begin
  move(st[bx+4], bx)           $   i := 0;
  move(name, cx)               $   while i < namelength do
do loadname2:                  $     i := i + 1;
  move(st[s], ax)              $     value[i] := char(
  move(ax, st[bx])             $       st[addr + i - 1])
  add(2, s)                    $   end
  add(2, bx)                   $ end
  loop(loadname2)
  pop(s)
  return(4)

do findname:                   $ proc findname(var id: name)
  move(st[t], bx)              $ var addr: int
  compare(1, st[tasks])        $ begin
  ifequal(findname2)           $   if tasks = 1 do
  move(st[progtop], bx)        $     addr := t + 1
```

```
do findname2:                 $   else true do
  add(2, bx)                  $     addr := progtop + 1
do findname3:                 $   end;
  compare(bx, p)              $   while addr < p do
  ifnothigher(findname4)      $     loadname(addr, id);
  push(bx)                    $     addr :=
  move(sp, bx)                $       addr + namelength;
  push(st[bx])                $     addr :=
  push(st[bx+4])              $       addr + st[addr]
  call(loadname)              $   end
  pop(bx)                     $ end
  add(namelength, bx)
  add(st[bx], bx)
  jump(findname3)
do findname4:
  return(2)

text linetext = ' line .'     $ * proc stop(lineno: int;
array idl [name]              $     reason: text)
do stop:                      $ var id: name
  move(idl, ax)               $ begin
  push(ax)                    $   findname(id);
  call(findname)              $   writechar(nl);
  move(nl, ax)                $   writename(id);
  push(ax)                    $   writetext(text(
  call(writechar)             $     ' line .'));
  move(idl, ax)               $   writeint(lineno);
  push(ax)                    $   writechar(sp);
  call(writename)             $   writetext(reason);
  move(linetext, ax)          $   writechar(nl);
  push(ax)                    $   beep;
  call(writetext)             $   "restart system"
  move(sp, bx)                $   kernel
  push(st[bx+4])              $ end
  call(writeint)
  move(space, ax)
  push(ax)
  call(writechar)
  add(2, sp)
  call(writetext)
  move(nl, ax)
  push(ax)
  call(writechar)
  call(beep)
  move(stackbottom, sp)
  move(false, st[blocked])
  move(1, st[this])
  move(1, st[tasks])
  move(st[b0], b)
  move(st[s0], s)
  move(st[t0], ax)
  move(ax, st[t])
  move(st[p0], p)
  jumpx(st[p])
```

```
text processtext =                $ * proc processlimit(
  'process limit exceeded.'       $     lineno: int)
do processlimit:                  $ begin
  pop(cx)                         $   stop(lineno, text(
  move(processtext, ax)           $     'process limit',
  push(ax)                        $     ' exceeded.'))
  call(stop)                      $ end

text variabletext =               $ * proc variablelimit(
  'variable limit exceeded.'      $     lineno: int)
do variablelimit:                 $ begin
  pop(cx)                         $   stop(lineno, text(
  move(variabletext, ax)          $     'variable limit',
  push(ax)                        $     ' exceeded.'))
  call(stop)                      $ end

text rangetext =                  $ * proc rangeerror(
  'range limit exceeded.'         $     lineno: int)
do rangeerror:                    $ begin
  pop(cx)                         $   stop(lineno, text(
  move(rangetext, ax)             $     'range limit',
  push(ax)                        $     ' exceeded.'))
  call(stop)                      $ end

text calltext =                   $ * proc callerror(
  'invalid program call.'         $     lineno: int)
do callerror:                     $ begin
  pop(cx)                         $   stop(lineno, text(
  move(calltext, ax)              $     'invalid program',
  push(ax)                        $     ' call.'))
  call(stop)                      $ end

text standardtext =               $ * proc standarderror
  'invalid standard call.'        $ begin
do standarderror:                 $   stop(1, text(
  move(1, ax)                     $     'invalid standard',
  push(ax)                        $     ' call.'))
  move(standardtext, ax)          $ end
  push(ax)
  call(stop)

word lineno                       $ * proc dividetrap(
do dividetrap:                    $     lineno: int)
  push(st[lineno])                $ "called by processor only"
  call(rangeerror)                $ begin rangeerror(lineno)
                                  $ end

const dividevector = 0            $ begin
do failure:                       $   "set the divide trap"
  move(dividevector, bx)          $      skip
  move(dividetrap, ax)            $ end "program failure"
  move(ax, st[bx])
  move(0, st[bx+2])
  jump(begin)
```

```
text disktext =                $ post proc diskproblem
  '[10]Disk error, .'          $ begin
do diskproblem:                $   writetext(text(nl,
  move(disktext, ax)           $     'Disk error, .'));
  push(ax)                     $   beep;
  call(writetext)              $   pause
  call(beep)                   $ end
  call(pause)
  return
                               $ proc loadset(addr: int;
                               $   var value: settype1)
                               $ var i: int
                               $ begin i := 0;
                               $   while i < setlength do
                               $     i := i + 1;
                               $     value:settype2[i] :=
                               $       st[addr + i - 1]
                               $   end
                               $ end

                               $ proc storeset(addr: int;
                               $   value: settype1)
                               $ var i: int
                               $ begin i := 0;
                               $   while i < setlength do
                               $     i := i + 1;
                               $     st[addr + i - 1] :=
                               $       value:settype2[i]
                               $   end
                               $ end

do preempt:                    $ proc preempt
  move(st[this], bx)           $ begin
  decrement(bx)                $   q[this].bx := b;
  move(3, cx)                  $   q[this].sx := s;
  shiftleft(bx)                $   q[this].tx := t;
  add(q, bx)                   $   q[this].px := p
  move(b, st[bx])              $ end
  move(s, st[bx+2])
  move(st[t], ax)
  move(ax, st[bx+4])
  move(p, st[bx+6])
  return

do resume:                     $ proc resume
  move(st[this], bx)           $ begin
  decrement(bx)                $   b := q[this].bx;
  move(3, cx)                  $   s := q[this].sx;
  shiftleft(bx)                $   t := q[this].tx;
  add(q, bx)                   $   p := q[this].px
  move(st[bx], b)              $ end
  move(st[bx+2], s)
  move(st[bx+4], ax)
  move(ax, st[t])
```

```
   move(st[bx+6], p)
   return

do switch:                     $ proc switch
   call(preempt)               $ begin
   move(st[this], ax)          $   preempt;
   increment(ax)               $   this :=
   compare(ax, st[tasks])      $     this mod tasks + 1;
   ifnotless(switch2)          $   resume
   move(1, ax)                 $ end
do switch2:
   move(ax, st[this])
   call(resume)
   return

const maxcode = 24576          $ const maxcode= 12288 "12 K words"

do moveprogram:                $ proc moveprogram
   subtract(maxcode, s)        $ var m: int
   move(st[s+2], cx)           $ begin
   subtract(namelength, s)     $   m := st[s - maxcode + 1]
   add(namelength, cx)         $     + namelength;
   push(s)                     $   s := s - maxcode
   add(cx, s)                  $     - namelength;
   move(st[t], bx)             $   t := t - m;
   subtract(cx, st[t])         $   while m > 0 do
   halve(cx)                   $     st[t + m] := st[s + m];
do moveprogram2:               $     m := m - 1
   move(st[s], ax)             $   end
   move(ax, st[bx])            $ end
   subtract(2, s)
   subtract(2, bx)
   loop(moveprogram2)
   pop(s)
   return

const progsector = 12          $ const progsector = 12
                               $"set starting location of os."
word sectornol                 $ proc loadprogram
word addrl                     $ var sectorno, addr: int
do loadprogram:                $ begin
   move(s, st[addrl])          $   addr := s + 1;
   add(2, st[addrl])           $   s := s + namelength
   add(namelength, s)          $     + maxcode;
   add(maxcode, s)             $   sectorno := progsector;
   move(progsector,            $   while addr < s do
     st[sectornol])            $     diskio(input, drive,
do loadprogram2:               $       sectorno, addr);
   compare(st[addrl], s)       $     sectorno :=
   ifnothigher(loadprogram3)   $       sectorno + 1;
   move(input, ax)             $     addr :=
   push(ax)                    $       addr + sectorlength
   move(drive, ax)             $   end;
   push(ax)                    $   moveprogram
```

```
    push(st[sectornol])           $ end
    push(st[addrl])
    call(diskio)
    increment(st[sectornol])
    add(sectorlength, st[addrl])
    jump(loadprogram2)

  do loadprogram3:
    call(moveprogram)
    return
  do initialize:                  $ proc initialize
    move(false, st[blocked])      $ begin
    move(1, st[this])             $    blocked := false;
    move(1, st[tasks])            $    this := 1;
    move(minaddr, b)              $    tasks := 1;
    add(paramlength, b)           $    b := minaddr + paramlength;
    move(pdpll, st[b-58])         $*****************************
    move(maxrow, st[b-56])        $*   st[b - 29] := int(pdpll);*
    move(maxcolumn, st[b-54])     $*   st[b - 28] := maxrow;    *
    move(cursorno, st[b-50])      $*   st[b - 27] := maxcolumn; *
    move(eraseno, st[b-46])       $*   st[b - 25] := cursorno;  *
    move(displayno, st[b-42])     $*   st[b - 23] := eraseno;   *
    move(acceptno, st[b-38])      $*   st[b - 21] := displayno; *
    move(printno, st[b-34])       $*   st[b - 19] := acceptno;  *
    move(getno, st[b-30])         $*   st[b - 17] := printno;   *
    move(putno, st[b-26])         $*   st[b - 15] := getno;     *
    move(formatno, st[b-22])      $*   st[b - 13] := putno;     *
    move(initportno, st[b-18])    $*   st[b - 11] := formatno;  *
    move(sendportno, st[b-14])    $*   st[b - 9] := initportno; *
    move(recportno, st[b-10])     $*   st[b - 7] := sendportno; *
    move(testkeyno, st[b-6])      $*   st[b - 5] := recportno;  *
    move(testportno, st[b-2])     $*   st[b - 3] := testkeyno;  *
    move(b, s)                    $*   st[b - 1] := testportno; *
                                  $*****************************
    add(8, s)                     $    s := b + 4;
    move(none, st[s])             $    st[s] := none;
    move(maxaddr, st[t])          $      "dummy return address"
    call(loadprogram)             $    t := maxaddr;
    move(st[t], p)                $    loadprogram;
    add(namelength, p)            $    p := t + namelength + 2
    add(4, p)                     $ end
    move(b, st[b0])
    move(s, st[s0])
    move(st[t], ax)
    move(ax, st[t0])
    move(p, st[p0])
    return

  $ "Procedure parameters"

  do cursor:                      $ proc cursor
    subtract(4, s)                $ var row, column: int
    push(st[s+2])                 $ begin s := s - 2;
    push(st[s+4])                 $    row := st[s + 1];
  call(putcursor)              $   column := st[s + 2];
  return                       $   putcursor(row, column)
                               $ end

do erase:                      $ proc erase
  call(erasescreen)            $ begin erasescreen end
  return

do display:                    $ proc display
  push(st[s])                  $ var value: char
  call(write)                  $ begin value := char(st[s]);
  subtract(2, s)               $   write(value);
  return                       $   s := s - 1
                               $ end

do accept:                     $ proc accept
  push(st[s])                  $ var addr: int
  call(read)                   $ begin addr := st[s];
  subtract(2, s)               $   readx(st[addr]:char);
  return                       $   s := s - 1
                               $ end

do print:                      $ proc print
  push(st[s])                  $ var value: char
  call(printchar)              $ begin value := char(st[s]);
  subtract(2, s)               $   printchar(value);
  return                       $   s := s - 1
                               $ end

do get:                        $ proc get
  subtract(6, s)               $ var driveno, sectorno,
  move(input, ax)              $   addr: int
  push(ax)                     $ begin s := s - 3;
  push(st[s+2])                $   driveno := st[s + 1];
  push(st[s+4])                $   sectorno := st[s + 2];
  push(st[s+6])                $   addr := st[s + 3];
  call(diskio)                 $   diskio(input, driveno,
  return                       $     sectorno, addr)
                               $ end

do put:                        $ proc put
  subtract(6, s)               $ var driveno, sectorno,
  move(output, ax)             $   addr: int
  push(ax)                     $ begin s := s - 3;
  push(st[s+2])                $   driveno := st[s + 1];
  push(st[s+4])                $   sectorno := st[s + 2];
  push(st[s+6])                $   addr := st[s + 3];
  call(diskio)                 $   diskio(output, driveno,
  return                       $     sectorno, addr)
                               $ end

do format:                     $ proc format
  subtract(4, s)               $ var driveno, trackno: int
  push(st[s+2])                $ begin s := s - 2;
```

```
    push(st[s+4])                   $   driveno := st[s + 1];
    call(formattrack)               $   trackno := st[s + 2];
    return                          $   formattrack(driveno, trackno)
                                    $ end
                                    $***********************
  do initport:                      $ proc testport        *
    push(st[s])                     $ var value: int       *
    call(initportl)                 $ begin value := st[s];*
    subtract(2,s)                   $   initportl(value);  *
    return                          $   s := s - 1         *
                                    $ end                  *
                                    $***********************

                                    $*****************************
  do sendport:                      $ proc sendport              *
    push(st[s])                     $ var value: int             *
    call(sendportl)                 $ begin value := char(st[s]);*
    subtract(2,s)                   $   sendportl(value);        *
    return                          $   s := s - 1               *
                                    $ end                        *
                                    $*****************************
  do recport:                       $ proc recport         *
    push(st[s])                     $ var addr: int        *
    call(recportl)                  $ begin addr := st[s]; *
    subtract(2,s)                   $   recportl(st[addr]);*
    return                          $   s := s - 1         *
                                    $ end                  *
                                    $***********************
  do testkey:                       $ proc testkey         *
    push(st[s])                     $ var addr: int        *
    call(testkeyl)                  $ begin addr := st[s]; *
    subtract(2,s)                   $   testkeyl(st[addr]);*
    return                          $   s := s - 1         *
                                    $ end                  *
                                    $************************
  do testport:                      $ proc testport         *
    push(st[s])                     $ var addr: int         *
    call(testportl)                 $ begin addr := st[s];  *
    subtract(2,s)                   $   testportl(st[addr]);*
    return                          $   s := s - 1          *
                                    $ end                   *
                                    $************************
  do feasible:                      $ proc feasible(
    move(sp, bx)                    $   procno: int): bool
    move(true, st[bx+4])            $ "Determines whether a given
    compare(acceptno, st[bx+2])     $  kernelcall    can       be
    ifnotequal(feasible2)           $  completed  without  delay"
    push(cx)                        $ begin
    call(ready)                     $   if procno = acceptno do
    pop(cx)                         $     val feasible := ready
    move(sp, bx)                    $   else true do
    move(cx, st[bx+4])              $     val feasible := true
  do feasible2:                     $   end
    return(2)                       $ end
```

```
do kernelcall:                      $ proc kernelcall(
  move(sp, bx)                      $   procno: int)
  move(st[bx+2], ax)                $ begin
  move(kernelcall13, cx)            $   if procno = cursorno
  push(cx)                          $     do cursor
  compare(cursorno, ax)             $   else procno = eraseno
  ifnotequal(kernelcall2)           $     do erase
  jump(cursor)                      $   else procno = displayno
do kernelcall2:                     $     do display
  compare(eraseno, ax)              $   else procno = acceptno
  ifnotequal(kernelcall3)           $     do accept
  jump(erase)                       $   else procno = printno
                                    $     do print

do kernelcall3:
  compare(displayno, ax)            $   else procno = getno
  ifnotequal(kernelcall4)           $     do get
  jump(display)                     $   else procno = putno
do kernelcall4:                     $     do put
  compare(acceptno, ax)             $   else procno = formatno
  ifnotequal(kernelcall5)           $     do format
  jump(accept)
                                    $****************************
                                    $   else procno = initportno*
do kernelcall5:                     $      do initport          *
  compare(printno, ax)              $   else procno = sendportno*
  ifnotequal(kernelcall6)           $     do sendport           *
  jump(print)                       $   else procno = recportno *
do kernelcall6:                     $     do recport            *
  compare(getno, ax)                $   else procno = testkeyno *
  ifnotequal(kernelcall7)           $     do testkey            *
  jump(get)                         $   else procno = testportno*
do kernelcall7:                     $     do testport           *
                                    $****************************
  compare(putno, ax)                $   end
  ifnotequal(kernelcall8)           $ end
  jump(put)
do kernelcall8:
  compare(formatno, ax)
  ifnotequal(kernelcall9)
  jump(format)
do kernelcall9:                     $ added t scott 1/1985
  compare(initportno, ax)
  ifnotequal(kernelcall10)
  jump(initport)
do kernelcall10:
  compare(sendportno, ax)
  ifnotequal(kernelcall11)
  jump(sendport)
do kernelcall11:
  compare(recportno, ax)
  ifnotequal(kernelcall12)
  jump(recport)
do kernelcall12:
  compare(testkeyno, ax)
  ifnotequal(kernelcall13)
  jump(testkey)
do kernelcall13:
  compare(testportno,ax)
  ifnotequal(kernelcall14)
  jump(testport)
do kernelcall14:                $ 9 changed to 14 by t scott 1/1985
  return(2)

$ "Standard instructions"

$ "Comment:
$    Empty # 'newline'"

do newline:                     $ proc newline(lineno: int)
  add(4, p)                     $ begin p := p + 2 end
  jumpx(st[p])

$ "Library procedure:
$    'goto' 'libproc' Expression 'endlib'"

do goto:                        $ proc goto(displ: int)
  add(st[p+2], p)               $ begin p := p + displ end
  jumpx(st[p])

do libproc:                     $ proc libproc(paramlength,
  compare(1, st[tasks])         $   templength, lineno: int)
  ifequal(libproc2)             $ begin
  push(st[p+6])                 $   if tasks > 1 do
  call(callerror)               $     callerror(lineno)
do libproc2:                    $   end;
  move(b, ax)                   $   st[b + 2] :=
  subtract(st[p+2], ax)         $     b - paramlength - 1;
  subtract(2, ax)               $   if s + templength > t do
  move(ax, st[b+4])             $     variablelimit(lineno)
  move(s, ax)                   $   end;
  add(st[p+4], ax)              $   p := p + 4
  compare(st[t], ax)            $ end
  ifnothigher(libproc3)
  push(st[p+6])
  call(variablelimit)
do libproc3:
  add(8, p)
  jumpx(st[p])

do endlib:                      $ proc endlib(lineno: int)
  call(moveprogram)             $ begin
  move(st[t], p)                $   moveprogram;
  add(namelength, p)            $   p := t + namelength + 2
  add(4, p)                     $ end
  jumpx(st[p])

$ "Complete procedure:
```

```
$    [ 'goto' ] 'procedure' [ Declaration ]*
$      Statement part 'endproc'"

do procedure:                     $ proc procedure(
  move(b, ax)                     $   paramlength, varlength,
  subtract(st[p+2], ax)           $   templength, lineno: int)
  subtract(2, ax)                 $ begin
  move(ax, st[b+4])               $   st[b + 2] :=
  add(st[p+4], s)                 $     b - paramlength - 1;
  move(s, ax)                     $   s := s + varlength;
  add(st[p+6], ax)                $   if s + templength > t do
  compare(st[t], ax)              $     variablelimit(lineno)
  ifnothigher(procedure2)         $   end;
  push(st[p+8])                   $   p := p + 5
  call(variablelimit)             $ end
do procedure2:
  add(10, p)
  jumpx(st[p])

do endproc:                       $ proc endproc
  move(st[b+8], ax)               $ begin
  compare(none, ax)               $   if st[b+4] <> none do
  ifequal(endproc2)               $     p := st[b + 4];
  move(ax, p)                     $     t := st[b + 3];
  move(st[b+6], ax)               $     s := st[b + 2];
  move(ax, st[t])                 $     b := st[b + 1]
  move(st[b+4], s)                $   else true do
  move(st[b+2], b)                $     p := p + 1
  jumpx(st[p])                    $   end
do endproc2:                      $ end
  add(2, p)
  jumpx(st[p])

$ "Procedure declaration:
$    Complete procedure # Library procedure # Empty
$  Module declaration:
$    [ Declaration ]* Statement part
$  Declaration:
$    Procedure declaration # Module declaration # Empty"
```

APPENDIX E

This appendix has the operating system code for the Edison-Kermit system.

```
"The Edison-PC Operating system
             17 July 1982
 Copyright (c) 1982 Per Brinch Hansen"
"Changes 1985 by Terry Scott"

array sector [1:256] (int)

proc system(
  pdp11: bool;
  maxrow, maxcolumn: int;
  proc cursor(row, column: int);
  proc erase;
  proc display(value: char);
  proc accept(var value: char);
  proc print(value: char);
  proc read_sector(drive, sectorno: int;
    var value: sector);
  proc write_sector(drive, sectorno: int;
    var value: sector);
  proc format_track(drive, trackno: int);
  proc init_port(value: int);"configures serial port"
  proc send_port(value: char);"sends character to COM1 port"
  proc port_rec(var value: int);"rtns integer from COM1 port"
  proc test_key(var value: int);"rtns 1 if key pressed else 0"
  proc test_port(var value: int))"rtns port status in value "

const nl = char(10); sp = ' '


module "character sets"

* set charset (char)
* var capitals, comment: charset

* proc subset(first, last: char): charset
  var c: char; value: charset
  begin c := first; value := charset;
    while c <= last do
      value := value + charset(c);
      c := char(int(c) + 1)
    end;
    val subset := value
  end
```

```
* proc lowercase(c: char): char
  begin
    if c in capitals do
      c := char(int(c) + 32)
    end;
    val lowercase := c
  end

begin capitals := subset('A', 'Z');
  comment := charset(nl, sp)
end


module "integers"

* const minint = #100000; maxint = 32767
  var signs, digits, numeric: charset

* proc read_int(proc read(var c: char);
    var value: int)
  var c: char; plus: bool; digit: int
  begin value := 0; read(c);
    while c in comment do read(c) end;
    if c in signs do plus := c = '+'; read(c)
    else true do plus := true end;
    while c in digits do
      digit := int(c) - int('0');
      if value >= (minint + digit) div 10 do
        value := 10 * value - digit
      end;
      read(c)
    end;
    if plus and (value > minint) do
      value := - value
    end
  end

* proc write_int(proc write(c: char); value, length: int)
  const max = 6
  array numeral [1:max] (char)
  var no: numeral; min, i: int; negative: bool
  begin
    if value = minint do
      no := numeral('-32768'); min := max
    else value = 0 do
      no[max] := '0'; min := 1
    else true do
      if value < 0 do
        negative := true; value := - value
      else true do negative := false end;
      min := 0;
      while value > 0 do
        no[max - min] := char(value mod 10 + int('0'));
        min := min + 1; value := value div 10
      end;
      if negative do
        no[max - min] := '-'; min := min + 1
      end
    end;
    while length > min do
      write(sp); length := length - 1
    end;
    while min > 0 do
      min := min - 1; write(no[max - min])
    end
  end

begin signs := charset('+-');
  digits := subset('0', '9')
end


module "names"

 * const namelength = 12
 * array name [1:namelength] (char)
  var alphanum, letters: charset

* proc read_name(proc read(var c: char);
    var value: name)
  var i: int; c: char
  begin value := name(sp); read(c);
    while c in comment do read(c) end;
    if c in letters do
      i := 1;
      while (c in alphanum) and (i <= namelength) do
        c := lowercase(c); value[i] := c;
        read(c); i := i + 1
      end
    end
  end

* proc write_name(proc write(c: char); value: name)
  var i: int
  begin i := 1;
    while i <= namelength do
      write(value[i]); i := i + 1
    end
  end

* proc less_name(x, y: name): bool
  var i: int
  begin i := 1;
    while (i < namelength) and (x[i] = y[i]) do
      i := i + 1
    end;
    val less_name := x[i] < y[i]
  end
```

```
begin letters := capitals + subset('a', 'z');
  alphanum := letters + subset('0', '9') + charset('_')
end


module "lines"

 * const linelength = 80
 * array line [1:linelength] (char)

  var endline: charset

* proc write_line(proc write(c: char); text: line)
  var i: int; c: char
  begin i := 1; c := text[1];
    while not (c in endline) and (i < linelength) do
      write(c); i := i + 1; c := text[i]
    end;
    if c = nl do write(nl) end
  end

begin endline := charset(nl, '.') end


module "terminal"

* const bel = char(7); bs = char(8); ht = char(9);
    lf = char(10); cr = char(13); esc = char(27);
    del = char(127); right = ht; left = bs;
    tab = 5 "char"
  var normal: bool; graphic: charset;
    text: line; typed, used: int

* proc write_terminal(value: char)
  begin
    if normal and (value = lf) do display(cr) end;
    display(value)
  end

* proc writename_terminal(value: name)
  begin write_name(write_terminal, value) end

* proc writeline_terminal(value: line)
  begin write_line(write_terminal, value) end

  proc typeline
  var i, x, n: int; c: char
  begin text[1] := nl; n := 1; x := 1; accept(c);
    while c <> cr do
      if (c = left) and (x > 1) do
        display(bs); x := x - 1
      else c = right do
        i := x;
        if x + tab < n do x := x + tab
        else x < n do x := n end;
        while i < x do display(text[i]); i := i + 1 end
      else (c = del) and (x < n) do
        n := n - 1; i := x;
        while i < n do
          text[i] := text[i + 1]; display(text[i]);
          i := i + 1
        end;
        text[n] := nl; display(sp); i := n + 1;
        while i > x do display(bs); i := i - 1 end
      else (c in graphic) and (n < linelength - 1) do
        n := n + 1; i := n;
        while i > x do
          text[i] := text[i - 1]; i := i - 1
        end;
        text[x] := c; x := x + 1;
        while i < n do
          display(text[i]); i := i + 1
        end;
        while i > x do display(bs); i := i - 1 end
      end;
      accept(c)
    end;
    write_terminal(nl); typed := n; used := 0
  end


* proc read_terminal(var value: char)
  begin
    if normal do
      if used = typed do typeline end;
      used := used + 1; value := text[used]
    else true do accept(value) end
  end

* proc readname_terminal(var value: name)
  begin read_name(read_terminal, value) end

* proc readint_terminal(var value: int)
  begin read_int(read_terminal, value) end

* proc select_terminal(standard: bool)
  begin normal := standard; used := typed; display(nl) end

* proc pause_terminal
  var value: char
  begin
    writeline_terminal(
      line('push return to continue', nl));
    accept(value)
  end

begin
  if linelength < maxcolumn do
```

```
    writeline_terminal(line('line limit.')); halt
  end;
  graphic := subset(char(32), char(126));
  select_terminal(true);
  writeline_terminal(
    line('        The Edison-Kermit System', nl));
  writeline_terminal(
    line('1982 Per Brinch Hansen; changes 1985 Terry Scott', nl))
end


module "failures"

* proc assume1(condition: bool; text: line)
  begin
    if not condition do
      writeline_terminal(text); halt
    end
  end

* proc assume2(condition: bool; title: name;
    text: line)
  begin
    if not condition do
      writename_terminal(title);
      write_terminal(sp);
      writeline_terminal(text); halt
    end
  end

begin skip end


module "disk pages"

* const pagelength = 512; pagesectors = 2
* array page [1:pagelength] (char)
  array overlay [1:pagesectors] (sector)

* proc read_page(drive, pageno: int; var block: page)
  var i: int
  begin i := 1;
    while i <= pagesectors do
      read_sector(drive,
        pagesectors * (pageno - 1) + i - 1,
        block:overlay[i]);
      i := i + 1
    end
  end

* proc write_page(drive, pageno: int; var block: page)
  var i: int
  begin i := 1;
    while i <= pagesectors do
```

```
      write_sector(drive,
        pagesectors * (pageno - 1) + i - 1,
        block:overlay[i]);
      i := i + 1
    end
  end

begin skip end



module "disk maps"

  const pages = 160 "per disk side";
    firstpage = 27; lastpage = 320;
    available = maxint; endlist = 0
  array list [firstpage:lastpage] (int)

* record diskmap (free, next: int; status: list)

* proc allpages(sides: int): int
  begin val allpages := pages * sides - firstpage + 1
  end

* proc empty_diskmap(var map: diskmap): int
  begin val empty_diskmap := endlist end


* proc extend_diskmap(var map: diskmap;
    var address: int)
  var elem, succ: int
  begin assumel(map.free > 0, line('disk limit.'));
    while map.status[map.next] <> available do
      if map.next < lastpage do map.next := map.next + 1
      else true do map.next := firstpage end
    end;
    if address = endlist do address := map.next
    else true do
      elem := address; succ := map.status[elem];
      while succ <> endlist do
        elem := succ; succ := map.status[elem]
      end;
      map.status[elem] := map.next
    end;
    map.status[map.next] := endlist;
    map.free := map.free - 1
  end

* proc discard_diskmap(var map: diskmap;
    address: int)
  var succ: int
  begin
    if address <> endlist do
      while address <> endlist do
```

```
        succ := map.status[address];
        map.status[address] := available;
        map.free := map.free + 1;
        address := succ
      end;
      map.next := firstpage
    end
  end

* proc address_diskmap(var map: diskmap;
    address, pageno: int): int
  var succ, p: int
  begin assumel(address <> endlist, line('file limit.'));
    succ := map.status[address]; p := 1;
    while (p < pageno) and (succ <> endlist) do
      address := succ; succ := map.status[address];
      p := p + 1
    end;
    assumel(p = pageno, line(' file limit.'));
    val address_diskmap := address
  end

begin skip end


record position (pages, words: int)
record attributes (address: int; length: position;
  protected: bool)


module "disk catalogs"
  const maxitem = 45
  record item (title: name; attr: attributes)
  array table [1:maxitem] (item)
* record diskcatalog (size: int; contents: table)

  proc locate(var catalog: diskcatalog; key: name;
    var index: int; var found: bool)
  begin
    if catalog.size = 0 do found := false
    else true do
      index := 1;
      while (catalog.contents[index].title <> key) and
        (index < catalog.size) do index := index + 1
      end;
      found := catalog.contents[index].title = key
    end
  end

* proc list_diskcatalog(var catalog: diskcatalog;
    sides: int; proc write(c: char))
  var index: int; entry: item; lengthx: position;
    used: int
  begin index := 1; used := 0; write(nl);
```

```
    while index <= catalog.size do
      entry := catalog.contents[index];
      lengthx := entry.attr.length;
      write_name(write, entry.title);
      if entry.attr.protected do
        write_line(write, line(' protected  .'))
      else true do
        write_line(write, line(' unprotected.'))
      end;
      write_int(write, lengthx.pages, 4);
      write_line(write, line(' pages.'));
      if (0 < lengthx.pages) and
        (lengthx.pages < 64) do
          write_int(write, pagelength *
            (lengthx.pages - 1) + lengthx.words, 7);
          write_line(write, line(' words.'))
      end;
      write(nl); used := used + lengthx.pages;
      if index mod (maxrow - 5) = 0 do
        pause_terminal
      end;
      index := index + 1
    end;
    write(nl);
    if sides = 1 do
      write_line(write, line('Single-sided disk:.'))
    else true do
      write_line(write, line('Double-sided disk:.'))
    end;
    write(nl); write_int(write, catalog.size, 5);
    write_line(write, line(' entries', nl));
    write_int(write, used, 5);
    write_line(write, line(' pages used', nl));
    write_int(write, allpages(sides) - used, 5);
    write_line(write, line(' pages available', nl))
  end

* proc include_diskcatalog(var catalog: diskcatalog;
    key: name; attr: attributes)
  var x, y: item; index: int
  begin assumel(catalog.size < maxitem,
      line('catalog full.'));
    x := item(key, attr); index := 1;
    while index <= catalog.size do
      y := catalog.contents[index];
      assume2(x.title <> y.title, key,
        line(' ambiguous.'));
      if less_name(x.title, y.title) do
        catalog.contents[index] := x; x := y
      end;
      index := index + 1
    end;
    catalog.size := catalog.size + 1;
    catalog.contents[catalog.size] := x
```

```
   end

 * proc search_diskcatalog(var catalog: diskcatalog;
     key: name; var value: attributes; var found: bool)
   var index: int
   begin locate(catalog, key, index, found);
     if found do value := catalog.contents[index].attr end
   end

 * proc change_diskcatalog(var catalog: diskcatalog;
     key: name; value: attributes)
   var index: int; found: bool
   begin locate(catalog, key, index, found);
     assume2(found, key, line(' unknown.'));
     catalog.contents[index].attr := value
   end

 * proc exclude_diskcatalog(var catalog: diskcatalog;
     key: name)
   var index: int; found: bool
   begin locate(catalog, key, index, found);
     assume2(found, key, line(' unknown.'));
     while index < catalog.size do
       catalog.contents[index] :=
         catalog.contents[index + 1];
       index := index + 1
     end;
     catalog.size := catalog.size - 1
   end

 begin skip end


 module "disk library"

   const tracks = 40 "per disk side"; firsttrack = 0;
     sectors = 320 "per disk side"; firstsector = 0;
     labelpage = 25 "and 26"
   array labeloverlay [1:2] (page)

   array filler [1:6] (int)

   record disklabel (sides: int; map: diskmap;
     catalog: diskcatalog; unused: filler)

   array labelpair [0:1] (disklabel)

   array boolpair [0:1] (bool)

   var labels: labelpair; original: boolpair

 * proc check(drive: int)
   begin
     assume1((drive = 0) or (drive = 1),
        line('drive no invalid.'))
    end

    proc flushdisk_library(drive: int)
    begin
      if original[drive] do
        write_page(drive, labelpage,
          labels[drive]:labeloverlay[1]);
        write_page(drive, labelpage + 1,
          labels[drive]:labeloverlay[2]);
        original[drive] := false
      end
    end

  * proc insertold_library(drive: int)
    begin
      read_page(drive, labelpage,
        labels[drive]:labeloverlay[1]);
      read_page(drive, labelpage + 1,
        labels[drive]:labeloverlay[2]);
      original[drive] := false
    end

  * proc flush_library
    begin flushdisk_library(0);
      flushdisk_library(1)
    end

  * proc list_library(drive: int; proc write(c: char))
    begin check(drive);
      list_diskcatalog(labels[drive].catalog,
        labels[drive].sides, write)
    end

  * proc delete_library(drive: int; title: name)
    var attr: attributes; found: bool
    begin check(drive);
      search_diskcatalog(labels[drive].catalog,
        title, attr, found);
      if found do
        assume2(not attr.protected, title,
          line(' protected.'));
        discard_diskmap(labels[drive].map, attr.address);
        exclude_diskcatalog(labels[drive].catalog, title);
        original[drive] := true
      end
    end

  * proc create_library(drive: int; title: name)
    begin check(drive);
      delete_library(drive, title);
      include_diskcatalog(labels[drive].catalog, title,
        attributes(empty_diskmap(labels[drive].map),
          position(0, 0), false));
    original[drive] := true
  end

* proc protect_library(drive: int; title: name;
    value: bool)
  var attr: attributes; found: bool
  begin check(drive);
    search_diskcatalog(labels[drive].catalog,
      title, attr, found);
    assume2(found, title, line(' unknown.'));
    attr.protected := value;
    change_diskcatalog(labels[drive].catalog,
      title, attr);
    original[drive] := true
  end

* proc rename_library(drive: int; old, new: name)
  var attr: attributes; found: bool
  begin check(drive);
    search_diskcatalog(labels[drive].catalog,
      new, attr, found);
    assume2(not found, new, line(' ambiguous.'));
    search_diskcatalog(labels[drive].catalog,
      old, attr, found);
    assume2(found, old, line(' unknown.'));
    assume2(not attr.protected, old, line(' protected.'));
    exclude_diskcatalog(labels[drive].catalog, old);
    include_diskcatalog(labels[drive].catalog, new, attr);
    original[drive] := true
  end

* proc change_library(drive: int; title: name;
    new: attributes)
  var old: attributes; found: bool
  begin check(drive);
    search_diskcatalog(labels[drive].catalog,
      title, old, found);
    assume2(found, title, line(' unknown.'));
    assume2(not old.protected, title, line(' protected.'));
    change_diskcatalog(labels[drive].catalog, title, new);
    original[drive] := true
  end

* proc search_library(var drive: int; title: name;
    var attr: attributes)
  var found: bool
  begin drive := 0;
    search_diskcatalog(labels[0].catalog,
      title, attr, found);
    if not found do
      drive := 1;
      search_diskcatalog(labels[1].catalog,
        title, attr, found)
    end;
    assume2(found, title, line(' unknown.'))
  end

* proc address_library(drive, start, pageno: int): int
  begin check(drive);
    val address_library :=
      address_diskmap(labels[drive].map, start, pageno)
  end

* proc extend_library(drive: int; var start: int)
  begin check(drive);
    extend_diskmap(labels[drive].map, start);
    original[drive] := true
  end

begin original := boolpair(false, false);
  insertold_library(0); insertold_library(1)
  "Both drives contain disks"
end


module "disk files"

* record diskfile (title: name; open, safe, changed: bool;
    drive, start: int; size: position)

* proc open_file(var file: diskfile; title: name;
    var size: position)
  var drive: int; attr: attributes
  begin search_library(drive, title, attr);
    size := attr.length;
    file := diskfile(title, true, attr.protected, false,
      drive, attr.address, size)
  end

* proc read_file(var file: diskfile; pageno: int;
    var block: page)
  begin assume1(file.open, line('file closed.'));
    assume2((1 <= pageno) and
      (pageno <= file.size.pages),
      file.title, line(' limit.'));
    read_page(file.drive, address_library(file.drive,
      file.start, pageno), block)
  end

* proc write_file(var file: diskfile; pageno: int;
    var block: page)
  begin assume1(file.open, line('file closed.'));
    assume2(not file.safe, file.title,
      line(' protected.'));
    assume2((1 <= pageno) and
      (pageno <= file.size.pages),
      file.title, line(' limit.'));
    write_page(file.drive, address_library(file.drive,
        file.start, pageno), block)
    end

  * proc extend_file(var file: diskfile;
      newpage: bool; newwords: int)
    begin assumel(file.open, line('file closed.'));
      assume2(not file.safe, file.title,
        line(' protected.'));
      if newpage do
        extend_library(file.drive, file.start);
        file.size.pages := file.size.pages + 1
      end;
      file.size.words := newwords; file.changed := true
    end

  * proc end_file(var file: diskfile)
    begin assumel(file.open, line('file closed.'));
      if file.changed do
        change_library(file.drive, file.title,
          attributes(file.start, file.size, file.safe))
      end;
      file.open := false
    end

begin skip end


"*****************************************************
 *This module does all the required operations on the*
 *sequence number and produces the eight different   *
 *packets used in the system.                        *
 *****************************************************"
module "packets"

  const pktlength = 94; timeout = 12
* array frametype[1:96](char) "used for forming packets"
var seq: int; "seq. num for packets "
    block: page "holds page of info ready to send to port"

"takes filled frame and returns the checksum character"
* proc checksum(frame: frametype): char
  var total, cnt: int; hi_bit_on: bool
  begin
    cnt := 1;
    total := 0;
    while cnt < int(frame[2]) - 31 do "totals ascii values"
      cnt := cnt + 1;                 "keeps low order byte"
      total := (total + int(frame[cnt])) mod 256
    end;
    hi_bit_on := false;
    if total div 128 = 1 do "removes bit 7 from low byte"
      hi_bit_on := true;    "records it was on"
      total := total - 128
    end;
    if total div 64 = 1 do "removes bit 6 and adds it to bit"
      total := total - 63  "position 0"
    end;
    if hi_bit_on do        "adds bit 7 to bit position 1"
      total := total + 2
    end;
    total := total mod 64;
    val checksum := char(total + 32)
  end

"seq. num. set to -1 because it is incremented before used"
* proc init_seq
  begin seq := -1 end

* proc inc_seq
  begin seq := (seq + 1) mod 64 end

* proc dec_seq
  begin seq := seq - 1;
    if seq = -1 do
      seq := 63
    end
  end

* proc rtn_seq: int
  begin val rtn_seq := seq end

"Fills in fields: SOH, len, seq. num., pkt type, and checksum."
  proc initframe(var frame: frametype; length: int; kind: char)
  begin
    frame[1] := char(1);
    frame[2] := char(length + 30);
    frame[3] := char(seq + 32);
    frame[4] := kind;
    frame[int(frame[2]) - 30] := checksum(frame)
  end

* proc startframe(var frame: frametype) "Start init. packet"
  begin inc_seq;
    frame[5] := char(32 + pktlength); "sets packet length"
    frame[6] := char(32 + timeout);"sets timeout in sec"
    initframe(frame, 7, 'S')
  end

* proc eofframe(var frame: frametype) "End file packet"
  begin inc_seq;
    initframe(frame, 5, 'Z')
  end

* proc eotframe(var frame: frametype) "End transmission packet"
  begin inc_seq;
    initframe(frame, 5, 'B')
  end
```

```
* proc ackframe(var frame: frametype) "Posit. acknowlegment pkt"
  begin initframe(frame, 5, 'Y') end

"Packet used to ack the start initialization packet"
* proc ackframel(var frame: frametype)
  begin
    frame[5] := char(32 + pktlength); "sets packet length"
    frame[6] := char(32 + timeout); "sets timeout period in secs"
    initframe(frame, 7, 'Y')
  end

* proc nakframe(var frame: frametype) "Negat. acknowlegment pkt"
  begin initframe(frame, 5, 'N') end

"File header packet.  Places f_name in data portion of packet"
* proc headerframe(var frame: frametype; fname: name)
  var cnt: int; let: char
  begin inc_seq;
    cnt := 1;
    let := fname[cnt];
    while (let <> sp) and (cnt <= 11) do
      frame[4 + cnt] := let;
      cnt := cnt + 1;
      let := fname[cnt]
    end;
    initframe(frame, 4 + cnt, 'F')
  end

  array inttype[1:2](int) "holds high and low bytes of integer"
"Constructs data packet."
* proc dataframe(var frame: frametype; block: page; var cnt: int;
                 length: int; textfile: bool)
  const maxint = 32767; maxpktlen = 92
  var fill, value, i: int; let: char; values: inttype;
      negative: bool
  begin inc_seq;
    fill := 4;
    while (fill < maxpktlen) and (cnt < length) do
      cnt := cnt + 1;
      let := block[cnt];
      value := int(block[cnt]);
      if value < 0 do "if bit 7 of high byte on then negative so"
        value := value + maxint + 1; "zero out this bit and"
        negative := true             "record it was on."
      else true do
        negative := false
      end;
      values[1] := value mod 256; "low byte placed in values[1]"
      if negative do  "high byte placed in values[2].  If bit 7"
        values[2] := value div 256 + 128 "was on, then turn on"
      else true do
        values[2] := value div 256
      end;
      if textfile do "text file treated byte at a time and"
```

```
          i := 1        "binary file bytes are treated in pairs."
        else true do
          i := 2
        end;
        while i >= 1 do
          fill := fill + 1;
          if values[i] > 127 do "characters 128 to 255 sent as is"
            frame[fill] := char(values[i])
          else (values[i] > 31) and (values[i] <> 127) do "chars 32"
            frame[fill] := char(values[i]); "to 126 sent as is."
            if char(values[i]) = '#' do "if '#' then add another"
              fill := fill + 1;          "one. character stuffing."
              frame[fill] := '#'
            end
          else true do             "If char. 0 to 31 or 127 prefix"
            frame[fill] := '#'; "with '#'"
            fill := fill + 1;
            if values[i] <= 31 do "If char less than 32, add 64"
              values[i] := values[i] + 64 "to make it printable"
            else true do          "If char. is 127 then subtract 64."
              values[i] := 63
            end;
            frame[fill] := char(values[i])
          end;
          i := i - 1
        end
      end;
      initframe(frame,fill + 1, 'D') "fill in control fields."
    end

begin skip end


"*****************************************************************
 *Elementary procs and functions used by higher level procs.    *
 *****************************************************************"
module "Kermit utilities"

* proc clearscreen "clears monitor's screen"
  const maxrow = 24
  var row: int
  begin row := maxrow;
    while row > 0 do
      cursor(row, 1);
      erase;
      row := row - 1
    end;
    cursor(2, 1)
  end

"Calls port_rec.  Converts low byte to a char, increments
 num if high byte is negative."
  proc receive(var num: int; var let: char)
  var value: int
  begin
    port_rec(value);
    let := char(value mod 256);
    if value < 0 do "value negative port has timed out."
      num := num + 1         "timeout period is 1 second."
    end
  end

"Returns 1 if serial port send register is empty, else 0."
  proc ready(var value: int)
  var cnt: int
  begin
    cnt := 0;
    while cnt < 16 do          "time wasting loop"
      cnt := cnt + 1
    end;
    test_port(value);
    if value < 0 do "takes absolute value of number in value"
      value := value * (-1)
    end;
    value := (value mod 16384) div 8192 "tests if bit 5 of "
  end "high byte is on.  It is on if send register is empty."

"Places char. in serial port send register when it is empty."
* proc send_char(let: char)
  var value: int
  begin
    ready(value);
    while value <> 1 do "check if serial port send register"
      ready(value)      "is empty"
    end;
    send_port(let)
  end

"Determines the packet length using 2nd character in
 packet and sends the packet to the serial port"
* proc sendframe(frame: frametype)
  var cnt: int
  begin cnt := 0;
    while cnt < int(frame[2]) - 30 do
      cnt := cnt + 1;
      send_char(frame[cnt])
    end
  end

"Reads entire packet from serial port.  It waits for SOH
 and reads rest of packet.  If SOH is received at any time
 procedure starts reading packet over."
* proc readframe(var recframe: frametype; var failed: bool)
  var times, cnt: int; let: char; frame: frametype
  begin times := 0;
    receive(times, let);
    while (int(let) <> 1) and (times < 25) do "wait 25 s for SOH"
      receive(times, let)
    end;
    if int(let) = 1 do
      times := 0;
      receive(times, let);
      recframe[2] := let;"read pkt length and place in 2nd cell"
      cnt := 2;
      while (int(let) <> 1) and (cnt < int(recframe[2]) - 30) and
                                                     (times < 25) do
        cnt := cnt + 1;      "continue to read characters"
        receive(times, let);
        recframe[cnt] := let
      else (int(let) = 1) and (times < 25) do
        receive(times, let); "resynch. if SOH is received"
        cnt := 2;
        recframe[2] := let
      end
    end;
    if times >= 25 do
      failed := true
    else true do
      failed := false
    end
  end

begin skip end


"***********************************************************
 *Contains procedures called or used by the send procedure.*
 ***********************************************************"
module "send and check frames"

var num_gd_frms, num_bd_frms: int

* proc initfrmcntsd
  begin num_gd_frms := 0; num_bd_frms := 0 end
"Checks ack packet's seq. num., checksum, and type field.
 Returns true if all are okay, else it returns false."
* proc checkframel(recframe: frametype): bool
  var seqchar: char
  begin seqchar := char(rtn_seq + 32);
    if (recframe[3] = seqchar) and (recframe[4] = 'Y') and
       (recframe[int(recframe[2]) - 30] = checksum(recframe)) do
       val checkframel := true
    else true do
       val checkframel := false
    end
  end

"Sends a pkt, calls readframe which returns a packet or failed
 is true, if not failed it checks the packet using checkframel,
 and increments num_bd_frms or num_gd_frms depending on the
 result.  Trys to send a packet 5 times before it gives up."
* proc sendandwait(frame: frametype; var failed: bool)
```

```
   var recframe: frametype; cnt: int; done, badframe: bool
   begin cnt := 0;
     done := false;
     while not done do
       badframe := false;
       sendframe(frame);
       readframe(recframe, failed);
       if failed do
         badframe := true
       else not checkframel(recframe) do
         badframe := true
       end;
       if badframe do
         num_bd_frms := num_bd_frms + 1;
         cursor(4, 58);
         write_int(display, num_bd_frms, 4);
         cnt := cnt + 1
       else true do
         done := true;
         cursor(4, 31);
         num_gd_frms := num_gd_frms + 1;
         write_int(display, num_gd_frms, 4)
       end;
       if cnt = 6 do
         done := true;
         failed := true
       end
     end
   end

"Receives a disk page of info, calls dataframe so a data packet
 is formed, calls sendandwait to send frame and check for ack."
* proc sendpage(block: page; size: int; var failed: bool;
                 textfile: bool)
   var frame: frametype; cnt: int
   begin cnt := 0;
     failed := false;
     while (cnt < size) and not failed do
       dataframe(frame, block, cnt, size, textfile);
       sendandwait(frame, failed)
     end
   end

begin skip end


"**********************************************************
 *Contains procedures called or used by receive procedure.*
 **********************************************************"

module "receive and check frames"

* array dblebuftype[0:1](page)"accepts incoming info. Once page
 is full start on other page.  Between pkts it writes out page"
```

```
var num_gd_frms, num_bd_frms: int

* proc initfrmcntrc
  begin num_gd_frms := 0; num_bd_frms := 0 end

"Receives the file header frame and retrieves the file name from
 the data portion of the packet."
* proc getname(frame: frametype; var f_name: name)
  var cnt: int
  begin cnt := 4;
    while (cnt < int(frame[2]) - 31) and (cnt < 16) do
      cnt := cnt + 1;
      f_name[cnt - 4] := frame[cnt]
    end;
    while cnt < 16 do  "fills in remainder of f_name with blanks"
      cnt := cnt + 1;
      f_name[cnt - 4] := sp
    end
  end

"Function returns integer to recandcheck: 0 if okay and more data
 to come, 1 if okay and eof, 2 if previous packet received but
 otherwise okay, and 3 if any error."
  proc checkframe2(recframe: frametype; kind: char): int
  var seqchar : char
  begin
    seqchar := char(rtn_seq + 32);
    if(recframe[3] = seqchar) and
      (recframe[int(recframe[2]) - 30] = checksum(recframe)) do
      if recframe[4] = kind do
        val checkframe2 := 0
      else recframe[4] = 'Z' do
        val checkframe2 := 1
      end
    else recframe[3] = char(int(seqchar) - 1) do
      val checkframe2 := 2
    else true do
      val checkframe2 := 3
    end
  end

"Calls readframe and checkframe2.  If checkframe2 gives 0  or 1
 inc. num. good frames and rtns. pkt in recframe and if 1 sets
 quit to true, if 2 dec seq num, send ack, and read next pkt, if
 3 send nak and read next pkt.  If 2 or 3 inc num bad frames."
* proc recandcheck(var recframe: frametype; kind: char;
                   var quit, failed: bool)
  var frame: frametype; cnt, result: int
  begin inc_seq;
    quit := false;
    readframe(recframe, failed);
    if not failed do
      result := checkframe2(recframe, kind);
      cnt := 0;
```

```
        while (result > 1) and not failed do
          if result = 3 do
            nakframe(frame);
            sendframe(frame)
          else result = 2 do
            dec_seq;
            ackframe(recframe);
            sendframe(recframe);
            inc_seq
          end;
          readframe(recframe, failed);
          if not failed do
            result := checkframe2(recframe, kind)
          end;
          num_bd_frms := num_bd_frms + 1;
          cursor(4, 62);
          write_int(display, num_bd_frms, 4)
        end
      end;
      num_gd_frms := num_gd_frms + 1;
      cursor(4, 35);
      write_int(display, num_gd_frms, 4);
      if result = 1 do quit := true end
    end

  "Called by writeoutfile.  Proc rcvs data pkt, decodes values,
   and places them in dblebuf.  Once a page of dblebuf is filled
   it sets full to true and continues to write to other page."
    proc loadpagebuf(var dblebuf: dblebuftype; frame: frametype;
      var full: bool; var row, col, i, sum: int; textfile: bool)
    var cnt, value: int
    begin full := false;
      cnt := 4;
      while cnt < int(frame[2]) - 31 do
        if col >= 512 do "if page is full set full to true and"
          col := 0; "write info to other page in dblebuf."
          full := true;
          row := (row + 1) mod 2
        end;
        while (i >= 1) and (cnt < int(frame[2]) - 31)  do
          cnt := cnt + 1;
          if frame[cnt] <> '#' do"char not prefixed keep as is."
            value := int(frame[cnt])
          else true do  "if prefixed advance to next character."
            cnt := cnt + 1;
            if (frame[cnt] <> '#') do "if next char is not '#',"
              value := int(frame[cnt]);"subtract 64 unless it is"
              if ((value >= 64) and (value <= 95)) or"ascii value"
                  ((value >= 192) and (value <= 223)) do"191 or 63"
                value := value - 64
              else true do "if ascii value is 191 or 63 add 64"
                value := value + 64
              end
            else true do "if 2 '#' in a row, actual char is '#'."
```

```
              value := 35
            end
          end;
          if (value > 127) and (i = 2) do "if bit 7 of high byte"
            value := value - 256  "is on, make value negative."
          end;
          if i = 2 do
            sum := value * 256 "if first value of pair, convert to"
          else true do     "high byte value, else add to previous"
            value := sum + value "high byte value."
          end;
          i := i - 1
        end;
        if i = 0 do
          if textfile do "if text file only deal with one byte at"
            i := 1       "a time."
          else true do "if binary file deal with bytes in pairs."
            i := 2
          end;
          col := col + 1;
          dblebuf[row][col] := char(value) "place value in buffer"
        end
      end
    end

"Calls recandcheck which returns a pkt.  Assuming there are more
 data pkts and nothing failed, then loadpagebuf is called.  If
 loadpagebuf rtns true then buffer written to file and directory
 is updated.  An ack is sent.  This continues until eof pkt is
 received or failed becomes true.  Last partial page is written
 to the file, ack for eof is sent, eot is received and acked."
* proc writeoutfile(file: diskfile; var failed: bool; value:name)
    var frame, recframe: frametype; quit, full, textfile: bool;
     pageno, col, amount, row, i, sum : int; dblebuf: dblebuftype
    begin row := 0; col := 0;
      pageno := 0; sum := 0;
      recandcheck(recframe, 'D', quit, failed);
      if not failed do
        if value = name('text') do
          i := 1;
          textfile := true
        else true do
          i := 2;
          textfile := false
        end;
        loadpagebuf(dblebuf, recframe, full, row, col, i, sum,
                                                         textfile);
        while not quit and not failed do
          if full do
            pageno := pageno + 1;
            if full do
              amount := 512
            else true do
              amount := col
            end;
            extend_file(file, true, amount);
            write_file(file, pageno, dblebuf[(row + 1) mod 2])
          end;
          ackframe(frame);
          sendframe(frame);
          recandcheck(recframe, 'D', quit, failed);
          if not failed do
            loadpagebuf(dblebuf, recframe, full, row, col, i,
                                             sum, textfile)
          end
        end;
        if not failed do
          if full do
            amount := 512
          else true do
            amount := col
          end;
          extend_file(file, true, amount);
          write_file(file, pageno + 1, dblebuf[row]);
          ackframe(frame);
          sendframe(frame);
          end_file(file)
        end
      end
    end

  begin skip end


  "*******************************************************
   *Contains procedures called by the operating system   *
   *standard commands                                    *
   *******************************************************"

  module "standard commands"

  * proc readdrive(var drive: int)
    begin writeline_terminal(line('  Drive no = .'));
      readint_terminal(drive)
    end

  * proc readfile(var title: name)
    begin writeline_terminal(line('  File name = .'));
      readname_terminal(title)
    end

  * proc protect(value: bool)
    var drive: int; title: name
    begin readdrive(drive); readfile(title);
      protect_library(drive, title, value)
    end

  * proc list
  var drive: int
  begin readdrive(drive);
    list_library(drive, write_terminal)
  end

* proc delete
  var drive: int; title: name
  begin readdrive(drive); readfile(title);
    delete_library(drive, title)
  end

"Tests keyboard and if pressed, character sent to serial port.
 Char. obtained from serial port is displayed on monitor
 screen.  Escape from procedure by typing esc char., char(17)"
* proc connect
  const esc = char(17)
  var value, inval: int; outval: char
  begin
    send_port(cr);
    outval := ' ';
    while outval <> esc do
      test_key(value);
      if value = 1 do
        accept(outval);
        if outval <> esc do
          send_port(outval)
        end
      end;
      port_rec(inval);
      display(char(inval mod 128))
    end
  end

* proc rename
  var drive: int; old, new: name
  begin readdrive(drive);
    writeline_terminal(line('  Old name = .'));
    readname_terminal(old);
    writeline_terminal(line('  New name = .'));
    readname_terminal(new);
    rename_library(drive, old, new)
  end

"Obtains serial port configuration value from user and
 configures port by calling init_port kernel entry."
* proc init_portl
  var ok: char; value: int
  begin
    writeline_terminal
                  (line('Type t for default configuration.'));
    display(cr); display(nl);
    accept(ok);
    display(ok);display(cr); display(nl);
    if ok = 't' do
```

```
        value := 131
     else true do
        writeline_terminal(line('  Configuration value = .'));
        readint_terminal(value)
     end;
     init_port(value)
   end

   const n = 10
   array table[1:n](page)"accepts info from disk"
 "Sends series of pkts: start init., file header, data pkts,
  eof, eot.  Each time waits for ack before sending next pkt."
 * proc send
   var title, value: name; file: diskfile; size: position;
       block: table; pageno, lastfull, m, i: int;
       frame: frametype; textfile, failed: bool
   begin
     clearscreen;
     writeline_terminal(line('  File to be sent = .'));
     readname_terminal(title);
     writeline_terminal(line('  text or binary file = .'));
     readname_terminal(value);
     if value = name('text') do
       textfile := true
     else true do
       textfile := false
     end;
     writeline_terminal(line('  Packets sent successfully = .'));
     write_int(display, 0, 4);
     writeline_terminal(line('  Number of retries = .'));
     write_int(display, 0, 4);
     initfrmcntsd;
     open_file(file, title, size);
     init_seq;
     startframe(frame);
     sendandwait(frame, failed); "send start init packet"
     if not failed do
       headerframe(frame, title);
       sendandwait(frame, failed); "send file header packet"
       if (size.pages > 0) and not failed do
         pageno := 0; lastfull := size.pages - 1;
         while (pageno < lastfull) and not failed do
           m := 0;
           while (m < n) and (pageno + m < lastfull) do
             m := m + 1;
             read_file(file, pageno + m, block[m]) "load block"
           end;                                   "with disk info."
           i := 0;
           while (i < m) and not failed do
             i := i + 1; "send block info to port in data pkts."
             sendpage(block[i], pagelength, failed, textfile)
           end;
           pageno := pageno + m
         end;
```

```
          if not failed do "send last parital page to serial port"
            read_file(file, size.pages, block[1]);
            sendpage(block[i], size.words, failed, textfile)
          end
        end;
        if not failed do
          end_file(file);
          eofframe(frame);
          sendandwait(frame, failed); "send eof pkt"
          if not failed do
            eotframe(frame);
            sendandwait(frame, failed) "send eot pkt"
          end
        end
      end;
      cursor(6, 1);
      if not failed do
        writeline_terminal(line('  Send completed ok .'))
      else true do
        writeline_terminal(line('  Send failed .'))
      end;
      cursor(9, 1)
    end

"Receives in order the pkts: send init, file header, data, eof,
 eot.  It receives pkt, if okay sends ack otherwise sends nak."
* proc receive
    var title, temp, value: name; file: diskfile; size: position;
        block: table; drive: int; frame: frametype;
        quit, failed: bool
    begin clearscreen; init_seq;
      writeline_terminal(line('  text or binary file = .'));
      readname_terminal(value);
      writeline_terminal(line('  Receiving file drive = .'));
      readint_terminal(drive);
      writeline_terminal(
                 line('  Packets received successfully = .'));
      write_int(display, 0, 4);
      writeline_terminal(line('  Number of retries = .'));
      write_int(display, 0, 4);
      initfrmcntrc;
      temp := name('temp');
      recandcheck(frame, 'S', quit, failed);
      if not failed do
        ackframel(frame);
        sendframe(frame); "send ack for start init pkt"
        recandcheck(frame, 'F', quit, failed);
        if not failed do
          getname(frame, title); "retrieve file name from file"
          delete_library(drive, title); "header pkt and open"
          create_library(drive, temp); "file by that that name"
          open_file(file, temp, size);
          ackframe(frame);
          sendframe(frame);"ack file header frame. send file"
        writeoutfile(file, failed, value);"in data pkts"
        if not failed do
          rename_library(drive, temp, title);
          recandcheck(frame, 'B', quit, failed);
          if not failed do
            ackframe(frame);
            sendframe(frame);"send ack to eot pkt"
            cursor(6, 1);
            writeline_terminal(line('  Receive completed ok .'))
          end
        end
      end
    end;
    if failed do
      cursor(6, 1);
      writeline_terminal(line('  Receive failed .'))
    end;
    cursor(9, 1)
  end

begin skip end


"********************************************************
 * Main Program                                         *
 ********************************************************"

var op: name
begin
  while true do
    write_terminal(nl);
    writeline_terminal(line('Command = .'));
    readname_terminal(op);
    if op = name('connect') do connect
    else op = name('delete') do delete
    else op = name('initport') do init_portl
    else op = name('list') do list
    else op = name('protect') do protect(true)
    else op = name('receive') do receive
    else op = name('rename') do rename
    else op = name('send') do send
    else op = name('unprotect') do protect(false)
    else true do
      writeline_terminal(line('  Not a legal operation'));
      writeline_terminal(nl);
    end;
    flush_library
  end
end "Edison-Kermit Operating System"
```

APPENDIX F

This appendix contains the code for the Unix Kermit. This is the code found on the Interdata 3220 in the Kansas State University Department of Computer Science in the file /usrb/wb/kermit/kermit2.c, except for one small change found at the end of this code in the procedure xread.

```
/*
 *      K e r m i t  File Transfer Utility
 *
 *      UNIX Kermit, Columbia University, 1981, 1982, 1983
 *      Bill Catchings, Bob Cattani, Chris Maio, Frank da Cruz
 *
 *      usage: kermit [csr][dlbe line baud escapechar] [f1 f2 ...]
 *
 *      where c=connect, s=send [files], r=receive, d=debug,
 *      l=tty line, b=baud rate, e=escape char (decimal ascii code).
 *      For "host" mode Kermit, format is either "kermit r" to
 *      receive files, or "kermit s f1 f2 ..." to send f1 .. fn.
 *
 */

#define SYS3 0
#define V7 1
#define TRACE 1
#include <stdio.h>                      /* Standard UNIX definitions */
#if SYS3
#include <termio.h>
#endif
#if V7
#include <sgtty.h>
#endif
#include <signal.h>
#include <setjmp.h>

/* Conditional Compilation: 0 means don't compile it, nonzero means do */

#define UNIX            1       /* Conditional compilation for UNIX */
#define TOPS_20         0       /* Conditional compilation for TOPS-20 */
#define VAX_VMS         0       /* Ditto for VAX/VMS */
#define IBM_UTS         0       /* Ditto for Amdahl UTS on IBM systems */


/* Symbol Definitions */

#define MAXPACK         94      /* Maximum packet size */
#define SOH             1       /* Start of header */
#define SP              32      /* ASCII space */
#define CR              015     /* ASCII Carriage Return */
```

```
#define DEL            127     /* Delete (rubout) */
#define CTRLD          4
#define BRKCHR         CTRLD   /* Default escape character for CONNECT */

#define MAXTRY         5       /* Times to retry a packet */
#define MYQUOTE        '#'     /* Quote character I will use */
#define MYPAD          0       /* Number of padding characters I will need */
#define MYPCHAR        0       /* Padding character I need */
#define MYEOL          '\0    /* End-Of-Line character I need */
#define MYTIME         5       /* Seconds after which I should be timed out */
#define MAXTIM         20      /* Maximum timeout interval */
#define MINTIM         2       /* Minumum timeout interval */

#define TRUE           -1      /* Boolean constants */
#define FALSE          0


/* Global Variables */

int     size,                  /* Size of present data */
        n,                     /* Message number */
        rpsiz,                 /* Maximum receive packet size */
        spsiz,                 /* Maximum send packet size */
        pad,                   /* How much padding to send */
        timint,                /* Timeout for foreign host on sends */
        numtry,                /* Times this packet retried */
        oldtry,                /* Times previous packet retried */
        fd,                    /* File pointer of file to read/write */
        remfd,                 /* File pointer of the host's tty */
        image,                 /* -1 means 8-bit mode */
        remspd,                /* Speed of this tty */
        host,                  /* -1 means we're a host-mode kermit */
        debug;                 /* -1 means debugging */

char    state,                 /* Present state of the automaton */
        padchar,               /* Padding character to send */
        eol,                   /* End-Of-Line character to send */
        escchr,                /* Connect command escape character */
        quote,                 /* Quote character in incoming data */
        **filelist,            /* List of files to be sent */
        *filnam,               /* Current file name */
        recpkt[MAXPACK],       /* Receive packet buffer */
        packet[MAXPACK];       /* Packet buffer */

#if SYS3
struct termio
#endif
#if V7
struct sgttyb
#endif
        rawmode,               /* Host tty "raw" mode */
        cookedmode,            /* Host tty "normal" mode */
        remttymode;            /* Assigned tty line "raw" mode */
```

```
jmp_buf env;                    /* Environment ptr for timeout longjump */


/*
 *      m a i n
 *
 *      Main routine - parse command and options, set up the
 *      tty lines, and dispatch to the appropriate routine.
 */

main(argc,argv)
int argc;                               /* Character pointer for */
char **argv;                            /*  command line arguments */
{
  char *remtty,*cp;                     /* tty for CONNECT, char pointer */
  int speed, cflg, rflg, sflg;          /* speed of assigned tty, */
                                        /* flags for CONNECT, RECEIVE, SEND */
  if (argc < 2) usage();                /* Make sure there's a command line. */

  cp = *++argv; argv++; argc -= 2;      /* Set up pointers to args */

/* Initialize this side's SEND-INIT parameters */

  eol = CR;                             /* EOL for outgoing packets */
  quote = MYQUOTE;                      /* Standard control-quote char "#" */
  pad = 0;                              /* No padding */
  padchar = NULL;                       /* Use null if any padding wanted */

  speed = cflg = sflg = rflg = 0;       /* Turn off all parse flags */
  remtty = 0;                           /* Default is host (remote) mode */
  image = UNIX;                         /* Default to 8-bit mode for UNIX, */
                                        /*  7-bit CRLF mode for others */
  image = 0;                            /* KSU default to non-image always */
  escchr = BRKCHR;                      /* Default escape character */

  while ((*cp) != NULL)                 /* Get a character from the cmd line */
    switch (*cp++)                      /* Based on what the character is, */
     {                                  /*  do one of the folloing */
      case '-': break;                  /* Ignore dash (UNIX style) */
      case 'c': cflg++; break;          /* C = CONNECT command */
      case 's': sflg++; break;          /* S = SEND command */
      case 'r': rflg++; break;          /* R = RECEIVE command */
      case 'e': if (argc--)             /* E = specify escape char */
                  escchr = atoi(*argv++); /*  as ascii decimal number */
                else usage();
                if (debug) fprintf(stderr,"escape char is ascii %d0,escchr);
                break;
      case 'l': if (argc--)             /* L = specify tty line to use */
                  remtty = *argv++;
                else usage();
                if (debug) fprintf(stderr,"line %s0,remtty);
                break;
#if UNIX                                /* This part only for UNIX systems */
      case 'b': if (argc--) speed = atoi(*argv++); /* Set baud rate */
```

```
                else usage();
              if (debug) fprintf(stderr,"speed %d0,speed); break;

      case 'i': image = TRUE; break;     /* Image (8-bit) mode */
#endif /* UNIX */

      case 'd': debug = TRUE; break;     /* Debug mode */
     }

/* Done parsing */

  if ((cflg+sflg+rflg) != 1) usage();    /* Only one command allowed */

  remfd = 0;                             /* Start out as a host (remote) */
  host = TRUE;

  if (remtty)                            /* If another tty was specified, */
   {
    remfd = open(remtty,2);              /*  open it */
    if (remfd < 0)                       /*  check for failure */
     {
      fprintf(stderr,"Kermit: cannot open %s0,remtty);
      exit(-1);                          /*  Failed, quit. */
     }
    host = FALSE;                        /* Opened OK, flag local (not host) */
   }

/* Put the tty(s) into the correct modes */

#if SYS3
  ioctl(1,TCGETA,&cookedmode);
  ioctl(1,TCGETA,&rawmode);
  rawmode.c_iflag = 0;
  rawmode.c_oflag = 0;
  rawmode.c_cflag &= ~(CSIZE|PARENB|PARODD);
  rawmode.c_cflag |= CS8;
  rawmode.c_lflag = ISIG;
  rawmode.c_cc[4] = 1;  /* min chars */
  rawmode.c_cc[5] = 20;  /* 2.0 second timeout */
#endif

#if V7
  gtty(0,&cookedmode);                   /* Save current mode for later */
  gtty(0,&rawmode);
  rawmode.sg_flags |= (RAW|TANDEM);
  rawmode.sg_flags &= ~(ECHO|CRMOD);

  gtty(remfd,&remttymode);               /* If local kermit, get mode of */
                                         /*  assigned tty */
  remttymode.sg_flags |= (RAW|TANDEM);
  remttymode.sg_flags &= ~(ECHO|CRMOD);
#endif

#if UNIX && V7                           /* Speed changing for UNIX only */
```

```
  if (speed)                               /* User specified a speed? */
   {
    switch(speed)                          /* Get internal system code */
     {
        case 110: speed = B110; break;
        case 150: speed = B150; break;
        case 300: speed = B300; break;
        case 1200: speed = B1200; break;
        case 2400: speed = B2400; break;
        case 4800: speed = B4800; break;
        case 9600: speed = B9600; break;
        default: fprintf(stderr,"bad line speed0);
     }
    remttymode.sg_ispeed = speed;
    remttymode.sg_ospeed = speed;
   }
#endif /* UNIX */

  if (remfd) stty(remfd,&remttymode);     /* Put asg'd tty in raw mode */


/* All set up, now execute the command that was given. */

  if (cflg) connect();                     /* CONNECT command */

  if (sflg)                                /* SEND command */
   {
    if (argc--) filnam = *argv++;          /* Get file to send */
      else usage();
    filelist = argv;
#if SYS3
    if (host) ioctl(1,TCSETA,&rawmode);
#endif
#if V7
    if (host) stty(0,&rawmode);            /* Put tty in raw mode if remote */
#endif
    if (sendsw() == FALSE)                 /* Send the file(s) */
      printf("Send failed.0);            /* Report failure */
    else                                   /*  or */
      printf("OK0);                      /* success */
#if SYS3
    if (host) ioctl(1,TCSETA,&cookedmode);
#endif
#if V7
    if (host) stty(0,&cookedmode);         /* Restore tty */
#endif
   }

  if (rflg)                                /* RECEIVE command */
   {
#if SYS3
    if (host) ioctl(1,TCSETA,&rawmode);
#endif
#if V7
```

```
    if (host) stty(0,&rawmode);          /* Put tty in raw mode if remote */
#endif
    if (recsw() == FALSE)                /* Receive the file */
      printf("Receive failed.0);       /* Report failure */
    else                                 /*  or */
      printf("OK0);                    /* success */
#if SYS3
    if (host) ioctl(1,TCSETA,&cookedmode);
#endif
#if V7
    if (host) stty(0,&cookedmode);       /* Restore tty */
#endif
   }
 }

usage()                                  /* Give message if user makes */
{                                        /* a mistake in the command */
  fprintf(stderr,
     "usage: kermit [csr][di][lbe] [line] [baud] [esc char] [f1 f2 ...]0);
  exit();
}

/*
 *      s e n d s w
 *
 *      Sendsw is the state table switcher for sending
 *      files.  It loops until either it finishes, or
 *      an error is encountered.  The routines called by
 *      sendsw are responsible for changing the state.
 *
 */

sendsw()
{
 char sinit(),sfile(),seof(),sdata(),sbreak();

 state = 'S';                            /* Send initiate is the start state */
 n = 0;                                  /* Initialize message number */
 numtry = 0;                             /* Say no tries yet */
 while(TRUE)                             /* Do this as long as necessary */
  {
   switch(state)
    {
     case 'D':  state = sdata();  break; /* Data-Send state */
     case 'F':  state = sfile();  break; /* File-Send */
     case 'Z':  state = seof();   break; /* End-of-File */
     case 'S':  state = sinit();  break; /* Send-Init */
     case 'B':  state = sbreak(); break; /* Break-Send */
     case 'C':  return (TRUE);           /* Complete */
     case 'A':  return (FALSE);          /* "Abort" */
     default:   return (FALSE);          /* Unknown, fail */
    }
  }
}
```

```
/*
 *      s i n i t
 *
 *      Send Initiate: Send my parameters, get other side's back.
 */

char sinit()
{
  int num, len;                           /* Packet number, length */

  if (debug) fprintf(stderr,"sinit0);
  if (numtry++ > MAXTRY) return('A');    /* If too many tries, give up */
  spar(packet);                           /* Fill up with init info */
  if (debug) fprintf(stderr,"n = %d0,n);

#if UNIX
#if SYS3
  if (host)
    ioctl(1,TCFLSH,0);
  else
    ioctl(remfd,TCFLSH,0);
#endif
#if V7 && 0
  if (host)                               /* Clear any pending input */
    ioctl();                              /*  like stacked-up NAKs */
  else
    ioctl(remfd,TIOCFLUSH,0);
#endif
#endif /* UNIX */

  spack('S',n,6,packet);                  /* Send an S packet */
  switch(rpack(&len,&num,recpkt))         /* What was the reply? */
   {
    case 'N':  return(state);             /* NAK */

    case 'Y':                             /* ACK */
      if (n != num) return(state);        /* If wrong ACK, stay in S state */
      rpar(recpkt);                       /* Get other side's init info */
      if (eol == 0) eol = '0;            /* Check and set defaults */
      if (quote == 0) quote = '#';        /* Control-prefix quote */
      numtry = 0;                         /* Reset try counter */
      n = (n+1)%64;                       /* Bump packet count */
      if (debug) fprintf(stderr,"Opening %s0,filnam);
      fd = open(filnam,0);                /* Open the file to be sent */
      if (fd < 0) return('A');            /* if bad file descriptor, give up */
      if (!host) printf("Sending %s0,filnam);
      return('F');                        /* OK, switch state to F */

    case FALSE: return(state);            /* Receive failure, stay in S state */
    default: return('A');                 /* Anythig else, just "abort" */
   }
 }

/*
```

```
 *      s f i l e
 *
 *      Send File Header.
 */

char sfile()
{
  int num, len;                        /* Packet number, length */

  if (debug) fprintf(stderr,"sfile0);

  if (numtry++ > MAXTRY) return('A');  /* If too many tries, give up */
  for (len=0; filnam[len] != ' '; len++); /* Add up the length */

#if UNIX && 0                          /* Don't know why this is here */
  len++;                               /*  Add 1 */
#endif                                 /* But leaves an extra null */

  spack('F',n,len,filnam);             /* Send an F packet */
  switch(rpack(&len,&num,recpkt))      /* What was the reply? */
   {
    case 'N':                          /* NAK, just stay in this state, */
      if (n != (num=(--num<0)?63:num)) /*  unless NAK for next packet, */
        return(state);                 /*  which is just like an ACK */
                                       /*  for this packet, fall thru to... */
    case 'Y':                          /* ACK */
      if (n != num) return(state);     /* If wrong ACK, stay in F state */
      numtry = 0;                      /* Reset try counter */
      n = (n+1)%64;                    /* Bump packet count */
      size = bufill(packet);           /* Get first data from file */
      return('D');                     /* Switch state to D */

    case FALSE: return(state);         /* Receive failure, stay in F state */
    default:    return('A');           /* Something esle, just "abort" */
   }
}

/*
 *      s d a t a
 *
 *      Send File Data
 */

char sdata()
{
  int num, len;                        /* Packet number, length */

  if (numtry++ > MAXTRY) return('A');  /* If too many tries, give up */
  spack('D',n,size,packet);            /* Send a D packet */

  switch(rpack(&len,&num,recpkt))      /* What was the reply? */
   {
    case 'N':                          /* NAK, just stay in this state, */
      if (n != (num=(--num<0)?63:num)) /*  unless NAK for next packet, */
```

```
        return(state);                  /*  which is just like an ACK */
                                        /*  for this packet, fall thru to... */
     case 'Y':                          /* ACK */
       if (n != num) return(state);     /* If wrong ACK, fail */
       numtry = 0;                      /* Reset try counter */
       n = (n+1)%64;                    /* Bump packet count */
       if ((size = bufill(packet)) == EOF) /* Get data from file */
         return('Z');                   /* If EOF set state to that */
       return('D');                     /* Got data, stay in state D */

     case FALSE: return(state);         /* Receive failure, stay in D */
     default:    return('A');           /* Anything else, "abort" */
    }
}

/*
 *      s e o f
 *
 *      Send End-Of-File.
 */

char seof()
{
  int num, len;                         /* Packet number, length */
  if (debug) fprintf(stderr,"seof0);
  if (numtry++ > MAXTRY) return('A');   /* If too many tries, "abort" */
  spack('Z',n,0,packet);                /* Send a 'Z' packet */
  if (debug) fprintf(stderr,"seof1 ");
  switch(rpack(&len,&num,recpkt))       /* What was the reply? */
   {
     case 'N':                          /* NAK, fail */
       if (n != (num=(--num<0)?63:num)) /* ...unless for previous packet, */
         return(state);                 /*  in which case, fall thru to ... */
     case 'Y':                          /* ACK */
       if (debug) fprintf(stderr,"seof2 ");
       if (n != num) return(state);     /* If wrong ACK, hold out */
       numtry = 0;                      /* Reset try counter */
       n = (n+1)%64;                    /* and bump packet count */
       if (debug) fprintf(stderr,"closing %s, ",filnam);
       close(fd);                       /* Close the input file */
       if (debug) fprintf(stderr,"ok, getting next file0);
       if (gnxtfl() == FALSE)           /* No more files go? */
         return('B');                   /* if not, break, EOT, all done */
       if (debug) fprintf(stderr,"new file is %s0,filnam);
       fd = open(filnam,0);
       if (fd<0) return('A');
       if (!host) printf("Sending %s0,filnam);
       return('F');                     /* More files, switch state to F */

     case FALSE: return(state);         /* Receive failure, stay in state Z */
     default:    return('A');           /* Something else, "abort" */
    }
}
```

```
/*
 *      s b r e a k
 *
 *      Send Break (EOT)
 */

char sbreak()
{
  int num, len;                          /* Packet number, length */
  if (debug) fprintf(stderr,"sbreak0);
  if (numtry++ > MAXTRY) return('A');    /* If too many tries "abort" */

  spack('B',n,0,packet);                 /* Send a B packet */
  switch (rpack(&len,&num,recpkt))       /* What was the reply? */
   {
    case 'N':                            /* NAK, fail */
      if (n != (num=(--num<0)?63:num))   /* ...unless for previous packet, */
        return(state);                   /*  in which case, fall thru to ... */

    case 'Y':                            /* ACK */
      if (n != num) return(state);       /* If wrong ACK, fail */

      numtry = 0;                        /* Reset try counter */
      n = (n+1)%64;                      /* and bump packet count */
      return('C');                       /* switch state to Complete */

    case FALSE: return(state);           /* Receive failure, stay in state B */
    default:    return ('A');            /* Other, "abort" */
   }
}

/*
 *      r e c s w
 *
 *      This is the state table switcher for receiving files.
 */

recsw()
{
  char rinit(),rdata(),rfile();          /* Use these procedures */

  state = 'R';                           /* Receive is the start state */
  n = 0;                                 /* Initialize message number */
  numtry = 0;                            /* Say no tries yet */

  while(TRUE) switch(state)              /* Do until done */
   {
    case 'D':   state = rdata(); break;  /* Data receive state */
    case 'F':   state = rfile(); break;  /* File receive state */
    case 'R':   state = rinit(); break;  /* Send initiate state */
    case 'C':   return(TRUE);            /* Complete state */
    case 'A':   return(FALSE);           /* "Abort" state */
    }
  }
```

```
/*
 *      r i n i t
 *
 *      Receive Initialization
 */

char rinit()
{
  int len, num;                            /* Packet length, number */

  if (numtry++ > MAXTRY) return('A');      /* If too many tries, "abort" */
  switch(rpack(&len,&num,packet))          /* Get a packet */
  {
   case 'S':                               /* Send-Init */
     rpar(packet);                         /* Get the other side's init data */
     spar(packet);                         /* Fill up packet with my init info */
     spack('Y',n,6,packet);                /* ACK with my parameters */
     oldtry = numtry;                      /* Save old try count */
     numtry = 0;                           /* Start a new counter */
     n = (n+1)%64;                         /* Bump packet number, mod 64 */
     return('F');                          /* Enter File-Send state */

   case FALSE:  return (state);            /* Didn't get a packet, keep waiting */
   default:     return('A');               /* Some other packet type, "abort" */
  }
}

/*
 *      r f i l e
 *
 *      Receive File Header
 */

char rfile()
{
 int num, len;                             /* Packet number, length */

 if (numtry++ > MAXTRY) return('A');       /* "abort" if too many tries */
 switch(rpack(&len,&num,packet))           /* Get a packet */
  {
   case 'S':                               /* Send-Init, maybe our ACK lost */
     if (oldtry++ > MAXTRY) return('A'); /* If too many tries, "abort" */
     if (num == ((n==0)?63:n-1))           /* Previous packet, mod 64? */
      {                                    /* Yes, ACK it again */
       spar(packet);                       /* with our Send-Init parameters */
       spack('Y',num,6,packet);            /*  ... */
       numtry = 0;                         /* Reset try counter */
       return(state);                      /* Stay in this state */
      }
     else return('A');                     /* Not previous packet, "abort" */

   case 'Z':                               /* End-Of-File */
     if (oldtry++ > MAXTRY) return('A');
     if (num == ((n==0)?63:n-1))           /* Previous packet, mod 64? */
```

```
      {                                       /* Yes, ACK it again. */
       spack('Y',num,0,0);
       numtry = 0;
       return(state);                         /* Stay in this state */
      }
     else return('A');                        /* Not previous packet, "abort" */

   case 'F':                                  /* File Header, */
     if (num != n) return('A');               /* which is what we really want */
                                              /* The packet number must be right */
     if (!getfil(packet))                     /* Try to open a new file */
      {
       fprintf(stderr,"Could not create %s0); /* Giv up if can't */
       return('A');
      }
     else                                     /* OK, give message */
       if (!host) printf("Receiving %s0,packet);
     spack('Y',n,0,0);                        /* Acknowledge the file header */
     oldtry = numtry;                         /* Reset try counters */
     numtry = 0;                              /* ... */
     n = (n+1)%64;                            /* Bump packet number, mod 64 */
     return('D');                             /* Switch to Data state */

   case 'B':                                  /* Break transmission (EOT) */
     if (num != n) return ('A');              /* Need right packet number here */
     spack('Y',n,0,0);                        /* Say OK */
     return('C');                             /* Go to complete state */

   case FALSE: return(state);                 /* Couldn't get packet, keep trying */
   default:     return ('A');                 /* Some other packet, "abort" */
  }
}

/*
 *      r d a t a
 *
 *      Receive Data
 */

char rdata()
{
 int num, len;                                /* Packet number, length */

 if (numtry++ > MAXTRY) return('A');          /* "abort" if too many tries */
 switch(rpack(&len,&num,packet))              /* Get packet */
   case 'D':                                  /* Got Data packet */
{
    if (num != n)                             /* Right packet? */
     {                                        /* No */
      if (oldtry++ > MAXTRY) return('A'); /* If too many tries, give up */
      if (num == ((n==0)?63:n-1))             /* Else check packet number */
       {                                      /* Previous packet again? */
        spack('Y',num,6,packet);              /* Yes, re-ACK it */
        numtry = 0;                           /* Reset try counter */
```

```
          return(state);                    /* Stay in D, don't write out data! */
         }
        else return('A');                   /* sorry wrong number */
       }
                                            /* Got data with right packet number */
      bufemp(packet,fd,len);      .         /* Write the data to the file */
      spack('Y',n,0,0);                     /* Acknowledge the packet */
      oldtry = numtry;                      /* Reset the try counters */
      numtry = 0;                           /* ... */
      n = (n+1)%64;                         /* Bump packet number, mod 64 */
      return('D');                          /* Remain in data state */

    case 'F':                               /* Got a File Header */
      if (oldtry++ > MAXTRY) return('A'); /* If too many tries, "abort" */
      if (num == ((n==0)?63:n-1))           /* Else check packet number */
       {                                    /* It was the previous one */
        spack('Y',num,0,0);                 /* ACK it again */
        numtry = 0;                         /* Reset try counter */
        return(state);                      /* Stay in Data state */
       }
      else return('A');                     /* Not previous packet, "abort" */

    case 'Z':                               /* End-Of-File */
      if (num != n) return('A');            /* Must have right packet number */
      spack('Y',n,0,0);                     /* OK, ACK it. */
      close(fd);                  .         /* Close the file */
      n = (n+1)%64;                         /* Bump packet number */
      return('F');                          /* Go back to Receive File state */

    case FALSE:  return(state);             /* No packet came, keep waiting */
    default:     return('A');               /* Some other packet, "abort" */
   }
}

/*
 *      c o n n e c t
 *
 *      Establish a virtual terminal connection with the remote host, over an
 *      assigned tty line.
 *
 */

connect()
{
  int parent;                               /* Fork handle */
  char c = NULL, r = '7';

  if (host)              /* If in host mode, nothing to connect to */
   {
    fprintf(stderr,"Kermit: nothing to connect to0);
    return;
   }

  parent = fork();       /* Start fork to get typeout from remote host */
```

```
   if (parent)           /* Parent passes typein to remote host */
    {
     printf("Kermit: connected.70);    /* Give message */
#if SYS3
     ioctl(1,TCSETA,&rawmode);
#endif
#if V7
     stty(0,&rawmode);                      /* Put tty in raw mode */
#endif
     xread(0,&c,1);                          /* Get a character */
     while ((c&0x7f) != escchr)             /* Check for escape character */
      {                                     /* Not it */
       xwrite(remfd,&c,1);                   /* Write the character on screen */
       c = NULL;                            /* Nullify it */
       xread(0,&c,1);                        /* Get next character */
      }                                     /* Until escape character typed */
     kill(parent,9);                        /* Done, get rid of fork */
#if SYS3
     ioctl(1,TCSETA,&cookedmode);
#endif
#if V7
     stty(0,&cookedmode);                   /* Restore tty mode */
#endif
     printf("0ermit: disconnected.0); /* Give message */
     return;                                /* Done */
    }
  else                  /* Child does the reading from the remote host */
    {
     while(1)                               /* Do this forever */
      {
       xread(remfd,&c,1);
       xwrite(1,&c,1);
      }
    }
}

/*
 *      KERMIT utilities.
 */

clkint()                                    /* Timer interrupt handler */
{
 longjmp(env,TRUE);                         /* Tell rpack to give up */
}

/* tochar converts a control character to a printable one by adding a space */

char tochar(ch)
char ch;
{
  return(ch + ' ');               /* make sure not a control char */
}

/* unchar undoes tochar */
```

```
char unchar(ch)
char ch;
{
  return(ch - ' ');              /* restore char */
}

/*
 * ctl turns a control character into a printable character by toggling the
 * control bit (ie. ^A becomes A and A becomes ^A).
 */

char ctl(ch)
char ch;
{
 return(ch ^ 64);                /* toggle the control bit */
}

/*
 *      s p a c k
 *
 *      Send a Packet
 */

spack(type,num,len,data)
char type, *data;
int num, len;
{
  int i;                                /* Character loop counter */
  char chksum, buffer[100];             /* Checksum, packet buffer */
  register char *bufp;                  /* Buffer pointer */

  bufp = buffer;                        /* Set up buffer pointer */
  for (i=1; i<=pad; i++) xwrite(remfd,&padchar,1); /* Issue any padding */

  *bufp++ = SOH;                        /* Packet marker, ASCII 1 (SOH) */
  chksum = tochar(len+3);               /* Initialize the checksum */
  *bufp++ = tochar(len+3);              /* Send the character count */
  chksum = chksum + tochar(num);        /* Init checksum */
  *bufp++ = tochar(num);                /* Packet number */
  chksum = chksum + type;               /* Accumulate checksum */
  *bufp++ = type;                       /* Packet type */

  for (i=0; i<len; i++)                 /* Loop for all data characters */
   {
    *bufp++ = data[i];                  /* Get a character */
    chksum = chksum+data[i];            /* Accumulate checksum */
   }
  chksum = (chksum + (chksum & 192) / 64) & 63; /* Compute final checksum */
  *bufp++ = tochar(chksum);             /* Put it in the packet */
  *bufp = eol;                          /* Extra-packet line terminator */
  xwrite(remfd, buffer,bufp-buffer+1);   /* Send the packet */
  if (debug) putc('.',stderr);
}
```

```
/*
 *  r p a c k
 *
 *  Read a Packet
 *
 */

rpack(len,num,data)
int *len, *num;                         /* Packet length, number */
char *data;                             /* Packet data */
{
 int i, done;                           /* Data character number, loop exit */
 char chksum, t, type;                  /* Checksum, current char, pkt type */

#if UNIX                                /* TOPS-20 can't handle timeouts... */
 if (setjmp(env)) return FALSE;         /* Timed out, fail */
 signal(SIGALRM,clkint);                /* Setup the timeout */
 if ((timint > MAXTIM) || (timint < MINTIM)) timint = MYTIME;
 alarm(timint);

#if SYS3
 if (host)
   ioctl(1,TCFLSH,0);
 else
   ioctl(remfd,TCFLSH,0);
#endif
#if V7 && 0
 if (host)                              /* Clear any pending input, */
   ioctl();                             /*  like stacked-up ACKs */
 else
   ioctl(remfd,TIOCFLUSH,0);
#endif
#endif /* UNIX */

 while ((t&0x7f) != SOH) xread(remfd,&t,1);      /* Wait for packet header */

 done = FALSE;                          /* Got SOH, init loop */
 while (!done)                          /* Loop to get a packet */
  {
   xread(remfd,&t,1);                    /* Get character */
   if (!image) t &= 0177;               /* Handle parity */
   if ((t&0x7f) == SOH) continue;               /* Resynchronize if SOH */

   chksum = t;                          /* Start the checksum */
   *len = unchar(t)-3;                  /* Character count */

   xread(remfd,&t,1);                    /* Get character */
   if (!image) t &= 0177;               /* Handle parity */
   if (t == SOH) continue;              /* Resynchronize if SOH */
   chksum = chksum + t;                 /* Accumulate checksum */
   *num = unchar(t);                    /* Packet number */

   xread(remfd,&t,1);                    /* Get character */
   if (!image) t &= 0177;               /* Handle parity */
```

```
    if (t == SOH) continue;                 /* Resynchronize if SOH */
    chksum = chksum + t;                    /* Accumulate checksum */
    type = t;                               /* Packet type */

    for (i=0; i<*len; i++)                  /* The data itself, if any */
     {                                      /* Loop for character count */
      xread(remfd,&t,1);                     /* Get character */
      if (!image) t &= 0177;                /* Handle parity */
      if (t == SOH) continue;               /* Resynch if SOH */
      chksum = chksum + t;                  /* Accumulate checksum */
      data[i] = t;                          /* Put it in the data buffer */
     }
    data[*len] = 0;                         /* Mark the end of the data */

    xread(remfd,&t,1);                       /* Get last character (checksum) */

    if (!image) t &= 0177;                  /* Handle parity */
    if (t == SOH) continue;                 /* Resynchronize if SOH */
   done = TRUE;                             /* Got checksum, done */
  }
#if UNIX
 alarm(0);                                  /* Disable the timer interrupt */
#endif

 chksum = (chksum + (chksum & 192) / 64) & 63; /* Fold bits 7,8 into chksum */
 if (chksum != unchar(t)) return(FALSE); /* Check the checksum, fail if bad */

 return(type);                              /* All OK, return packet type */
}

/*
 *      b u f i l l
 *
 *      Get a bufferful of data from the file that's being sent.
 *      Only control-quoting is done; 8-bit & repeat count prefixes are
 *      not handled.
 */

bufill(buffer)
char buffer[];                              /* Buffer */
 {
  int i;                                    /* Loop index */
  char t,t7;
  i = 0;                                    /* Init data buffer pointer */
  while(read(fd,&t,1) > 0)                  /* Get the next character */
   {
    if (image)                              /* In 8-bit mode? */
     {
      t7 = t & 0177;                        /* Yes, look at low-order 7 bits */
      if (t7 < SP || t7==DEL || t7==quote) /* Control character? */
       {                                    /*    Yes, */
        buffer[i++] = quote;                /*    quote this character */
        if (t7 != quote) t = ctl(t);        /* and uncontrollify */
       }
      }
     else                                        /* Else, ASCII text mode */
      {
       t &= 0177;                                /* Strip off the parity bit */
       if (t < SP || t == DEL || t == quote) /* Control character? */
        {                                        /* Yes */
         if (t=='\n')                          /* If newline, squeeze CR in first */
          {
           buffer[i++] = quote;
           buffer[i++] = ctl('\r');
          }
         buffer[i++] = quote;                    /* Insert quote */
         if (t != quote) t=ctl(t);               /* Uncontrollified the character */
        }
      }
     buffer[i++] = t;                            /* Deposit the character itself */
     if (i >= spsiz-8) return(i);                /* Check length */
    }
   if (i==0) return(EOF);                        /* Wind up here only on EOF */
   return(i);                                    /* Handle partial buffer */
}

/*
 *      b u f e m p
 *
 *      Get data from an incoming packet into a file.
 */

bufemp(buffer,fd,len)
char buffer[];                                   /* Buffer */
int fd, len;                                     /* File pointer, length */
 {
  int i;                                         /* Counter */
  char t;                                        /* Character holder */

  for (i=0; i<len; i++)                          /* Loop thru the data field */
   {
    t = buffer[i];                               /* Get character */
    if (t == MYQUOTE)                            /* Control quote? */
     {                                           /* Yes */
      t = buffer[++i];                           /* Get the quoted character */
      if ((t & 0177) != MYQUOTE)                 /* Low order bits match quote char? */
        t = ctl(t);                              /* No, uncontrollify it */
     }
    if (image || (t != CR))                      /* Don't pass CR in text mode */
      write(fd,&t,1);                            /* Put the char in the file */
   }
 }

/*
 *      g e t f i l
 *
 *      Open a new file
 */
```

```
getfil(filenm)
char *filenm;
{
  if (filenm[0] == ' ')
    fd = creat(packet,0644);                /* If filename known, use it */
  else
    fd = creat(filenm,0644);                /* else use sourcefile name */
  return (fd > 0);                          /* Return false if file won't open */
}

/*
 *      g n x t f l
 *
 *      Get next file in a file group
 */

gnxtfl()
{
  if (debug) fprintf(stderr, "gnxtfl0);
  filnam = *(filelist++);
  if (filnam == 0) return FALSE;            /* If no more, fail */
  else return TRUE;                         /* else succeed */
}

/*
 *      s p a r
 *
 *      Fill the data array with my send-init parameters
 *
 */

spar(data)
char data[];
{
  data[0] = tochar(MAXPACK);                /* Biggest packet I can receive */
  data[1] = tochar(MYTIME);                 /* When I want to be timed out */
  data[2] = tochar(MYPAD);                  /* How much padding I need */
  data[3] = ctl(MYPCHAR);                   /* Padding character I want */
  data[4] = tochar(MYEOL);                  /* End-Of-Line character I want */
  data[5] = MYQUOTE;                        /* Control-Quote character I send */
}

/*      r p a r
 *
 *      Get the other host's send-init parameters
 *
 */

rpar(data)
char data[];
{
  spsiz = unchar(data[0]);                  /* Maximum send packet size */
  timint = unchar(data[1]);                 /* When I should time out */
  pad = unchar(data[2]);                    /* Number of pads to send */
  padchar = ctl(data[3]);                  /* Padding character to send */
  eol = unchar(data[4]);                   /* EOL character I must send */
  quote = data[5];                         /* Incoming data quote character */
}

xread (fd,buf,len)
int fd, len;
char *buf;
{
        int     i,n;
        n=read(fd,buf,len);
        for (i=0; i<n; i++)
                if(!image) buf[i] &= 0x7f; /* Changed by t. scott 5-1985 */
#if TRACE                                  /* if(!image) was added       */
        if (debug) {
                fprintf(stderr,"xread len=%x ",n);
                for (i=0; i<n; i++)
                        fprintf(stderr,"%x ",*(buf+i) & 0xff);
                fprintf(stderr,"0);
        };
#endif
        return n;
};

xwrite (fd,buf,len)
int fd, len;
char *buf;
{
        int     i,n;
        n=write(fd,buf,len);
#if TRACE
        if (debug) {
                fprintf(stderr,"xwrit len=%x ",n);
                for (i=0; i<n; i++)
                        fprintf(stderr,"%x ",*(buf+i) & 0xff);
                fprintf(stderr,"0);
        };
#endif
        return n;
};
```

BIBLIOGRAPHY

[Bou 1982] Bourne, S. R.: "The UNIX System", Addison_Wesley, 1982.

[Bri 1977] Brinch Hansen, Per: "The Architecture of Concurrent Programs", Prentice-Hall, Englewood Cliffs, 1977.

[Bri 1982] Brinch Hansen, Per: "Programming a Personal Computer", Prentice-Hall, Englewood Cliffs, 1982.

[Col 1983] MPC Asynchronous Communications Program: Perfect Link: Columbia Data Products, Columbia, Maryland, 1983

[Cru 1984A] Cruz, Frank da; Catchings, Bill: "Kermit: A File-Transfer Protocol for Universities. Part 1: Design Consideration and Specifications", Byte (June 1984) Vol. 9, No. 6, pp. 255-278.

[Cru 1984B] Cruz, Frank da; Catchings, Bill: "Kermit: A File-Transfer Protocol for Universities. Part 2: States and Transitions, Heuristic Rules, and Examples", Byte (July 1984) Vol. 9, No. 7, pp. 143-145, 400-403.

[Cru 1984C] Cruz, Frank da; Tzoar, Daphne; Catchings, Bill: "Kermit Users Guide", Columbia University Center for Computing Activities, New York, 1983.

[Dig 1975] "LSI11 PDP11/03 Processor Handbook": Digital Equipment Corporation, Maynard, MA., 1975.

[Hoa 1972] Hoare, C. A. R.: "Towards a Theory of Parallel Programming", Operating Systems Techniques, Academic Press, New York, pp. 61-71, 1972.

[Ker 1978] Kernighan, Brian W.; Ritchie, Dennis M.: The C Programming Language", Prentice-Hall, Englewood Cliffs, 1978.

[Nor 1983] Norton, Peter: "Inside the IBM PC", Brady, 1983, Bowie, MA, 1983.

[Wir 1971] Wirth, N.: "The programming language Pascal", Acta Informatica 1, pp. 35_63, 1971.

[Zen 1984] "MS-DOS Version 2 Programmer's Utility Pack": Zenith Data Systems Corporation, St. Joseph, MI, 1984, pp. 6.30-6.33.

An Implementation of the Kermit Protocol Using the Edison System

by

Terry A. Scott

B. S. Iowa State University, 1964

Ph. D. University Of Wyoming, 1972

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1985

ABSTRACT

This report is primarily concerned with the Edison-Kermit system. The Edison-Kermit system was developed to allow transfer of files between the Edison operating system running on the PC microcomputer and a minicomputer running Unix attached to the serial port of the microcomputer.

The Edison-Kermit system was created using the Edison operating system and program development environment hereafter called the Edison System. The Edison System was written by Per Brinch Hansen and consists of a relatively small and uncomplicated operating system, a compiler written for the Edison language, an editor, and several programs to assist in program development. All of the Edison System programs are written in the Edison language with the exception of the kernel of the operating system. This kernel is written in Alva, which is an assembly language patterned after PDP-11 assembly language.

The Kermit protocol is a sliding window protocol of size one. It uses a single character checksum and a sequence number to detect errors that might occur when files are transferred between the microcomputer and the minicomputer attached to the micro's serial port.

In addition to the discussion of the Edison-Kermit system, the report has chapters on the Edison language, the Edison operating system and program development environment, and

the Kermit file transfer protocol. There are also appendices which contain a user's manual, helpful hints for Edison System users, a Kermit session showing the packets transmitted by the sender and receiver, and the code for the Edison-Kermit operating system and kernel and the C code for the Kermit that runs on the minicomputer.