The Development of the Edison System as a Truly Portable Software Development Environment

by

Michael C. Wonderlich

B. S. Kansas State University, 1985

A Master's Report

submitted in partial fulfillment of the

requirements of the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1989

Approved by:

[illegible]
Major Professor

b8
4
SC
84
blo

Table of Contents

CHAPTER 1

INTRODUCTION

1.0 The Project

The Edison System [BH2] is a small operating system which was originally designed for microcomputer usage. Its primary goal was to provide a compact, simple operating system which could be easily understood and ported to new computer architectures. In an effort to reduce the amount of time and effort required to port this operating system, a project was developed to rewrite its underlying structure in a high level programming language. This approach relies on the ability of each architecture's operating system to service all privileged operations required by the Edison System.

The following report documents the progression and development of the methods used to implement Edison in the following computer environments: Unix with Berkeley 4.3 enhancements, Unix System V, Ultrix, and MS-DOS version 3.2. The accomplishment of implementing Edison on these systems demonstrates that writing all portions of the operating system in a high level language is a realistic option for system developers. The resulting operating system does not utilize system-dependent assembly language anywhere in the operating system. Edison has been tested in all of these computer environments.

1.1 Motivation for the project.

The Edison System is comprised of both an operating system and programming environment. The system language, also named Edison, is simple yet powerful. The Edison language is comprised of only a few basic commands and structures. However, these basic fundamentals have been carefully chosen to establish the necessary foundation to produce powerful and extensive applications. The Edison System possesses adequate development facilities, yet is contained in a very small quantity of code. Brinch Hansen's final development system contains approximately 10,000 lines of program text. This text includes the operating system, the compiler, the editor, the diskette formatting program, necessary assembly code, and other utility programs. This establishes a powerful yet small environment for the development of concurrent programs.

The original version of the Edison System can be transported to new computer environments with relative ease because all system-dependent features were combined into one section of code, namely, the Kernel. The Kernel is a collection of system dependent routines which control all critical functions of the operating system. For every new computer environment to which Edison is to be ported, the system Kernel must be rewritten in an assembly language acceptable to the host environment.

As a result of this project, a procedure was implemented which virtually eliminates the need to rewrite the Kernel. By writing the Kernel in a high-level language instead of assembly language, a system has been provided which is extremely adaptable to new computer environments. Instead of rewriting an entire Kernel, only small key areas will require revision. This completes a project previously begun by T. Scott for the Kansas State University Department of Computer and Information Science [SCO].

Scott developed a communication protocol implemented in Edison. This facilitated the movement of application software between Edison and other environments. The combination of these two projects allows an application program to be developed on any system and ported with no alterations to any other system which has Edison implemented on it. As a result of these projects, the Edison System and applications are available for use upon many different computer systems.

1.2 Background of the Edison System.

Brinch Hansen developed the Edison System with only a few primary goals [BH2]. Foremost among his goals was that the system should contain sufficient power to develop nontrivial programs. Also, the system was to be simple enough for a single programmer to understand its entirety. The system was to contain all necessary development resources to achieve the first two goals without the introduction of additional software. A final goal was that the system should be able to be easily ported to new computer environments.

To accomplish these goals, Brinch Hansen adopted some development rules. To aid in the reduction of complexity, the system was written in one language. A new language was developed for this purpose, Edison. The Edison language is very similar to Pascal, but with all of the complex data structures removed. The language only contains the basic commands and structures, from which all programming tasks may be accomplished. The only part of the Edison Development System which is not written in Edison is the Kernel, which was written in the Alva assembler language (the assembly language Alva is a formal notation in which programs can be written for the PDP 11 computers). The entire Edison System is contained in approximately 10,000 lines of code. This results in a compact operating system (only 1200 lines of Edison code) which occupies very little of the computer's memory, thus leaving more memory for the development programs to utilize.

The Edison programming language is patterned in a way that is similar to the style of Pascal. Some deletions, additions, and revisions were made to create a powerful, well-defined programming language. In addition to standard computation features found in modern programming languages, the Edison language also contains constructs for controlling the concurrent execution of tasks.

1.3 The organization of this report.

This report may be helpful to persons desiring information on: developing a program on the Edison System, studying a small compact operating system, the issues of portability and their relevance with operating system development, or the development of a Kernel in a high-level language.

Chapter Two provides the history of portable operating systems throughout the last two decades. Chapter Three provides a history and description of the Edison System. Chapter Four describes the need for the implementation of a high level language Kernel. Also, a description of the project is detailed in this chapter. Chapter Five describes the procedures taken to implement this project. For further details on the modified Kernel and operating system routines, please refer to the appendices included at the end of this report.

CHAPTER 2
HISTORY OF PORTABLE OPERATING SYSTEMS

2.0 History of Portable Operating Systems

The first operating systems were very limited. They can be best described as a monitor which supervised the execution of a single user's program. Such systems provided a method of beginning and halting the execution of the program. Also, during program execution such operating systems or monitors may have additionally provided procedures for programs to communicate with external devices to be accessed. To accomplish these tasks the monitors were written in machine language and were designed for a particular machine. Every new machine required extensive time and money to be invested in development of develop a new monitor. There was no standardization of user interfaces provided by the various monitors, which resulted in confusion for users and programs migrating from one system to another. Monitors grew in power and performance, but the lack of consistency continued.

2.1 IBM 360 operating system family

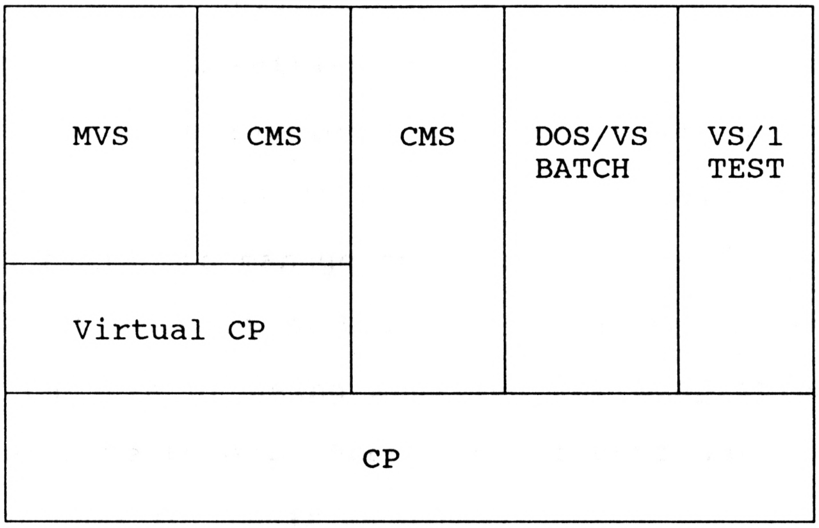
International Business Machines (IBM) developed the first major operating system with the capacity to span multiple processing units.[PET] Their desire was to establish a common operating system which would cover a family of computers. This was a noble attempt at a new design idea. A common environment would be established for everyone to work in and this would reduce the amount of time needed to provide the new software for various computer models. Thus only one set of application software was to be required for the entire family of computers. The reduced time and costs of design and maintenance were intended to greatly enhance IBM's position in the computer industry.

Unfortunately this bold attempt did not succeed as hoped. The operating system did provide a common operating system for a family of computers. The attempt to make a universal, powerful operating system, resulted in a product which was extremely large and confusing. It was impossible for any single person to understand the entire operating system. Over 17,000 hours of design and programming time went into the development of the IBM/360 operating system. While it was possible for the operating system to perform powerful functions, it was difficult to discover how to perform these functions.

Despite these problems the system was a success. By the standards of the time, great advances had been achieved with the 360 operating system design. The operating system was capable of being implemented upon several different models. Due to the complex projects desired, large software development teams were required to write application packages, but these applications were easier to develop and install than ever before. The improved efficiency of the 360 operating system provided IBM with the opportunity to concentrate more heavily in the development of new application software.

2.2 IBM VM operating system design.

The foundation of the 360 and the 370 operating system family is the concept of a virtual monitor.[IBM] The 360/370 system is designed in layers. At the lowest level is a control program (CP) which has the capability of managing multiple virtual machines simultaneously. A virtual machine is the logical representation of a desired machine. The resources it sees are actually constructs of the CP which provide all the appearances of the physical device. A CP maintains a mapping from the virtual devices to the physical devices. Each virtual machine may be executing a different operating system. The notion of a virtual monitor even provides the ability to test a new CP in a virtual machine which would then in turn provide another level of virtual machines (Figure 2.1). This design proved to be very flexible and provided great strengths for the development of software. Specifically, the ability to create a test environment to simulate the system the application software will be utilized in.



IBM Virtual Machine Model
Figure 2.1

2.3 Unix operating system.

The next major attempt at a universal operating system met with better luck. Unix was developed by Richie and Thompson in 1974.[PET] A new design idea was used which emphasized simplicity rather than speed or sophistication. A high-level language, C, was used to write a majority of the operating system code. C provided the constructs necessary to produce a multi-tasking operating system. It also follows the formal definitions of a structured language. These features allow software to be written in an efficient and easily readable fashion. The result was an operating system written in about 10,000 lines of code. This proved to be much easier to manage and transport to new architectures. With the advent of Unix, an operating system existed which was capable of being modified for adaptation to new computer architectures. Soon many vendors were producing versions of Unix for their machines.

2.2.1 The Kernel design concept.

The primary reason for the success in porting Unix between machines can be found in the structure of its Kernel. The Kernel is a library of routines to provide system level control. Whenever a process desires a function of the system, a request is made to the Kernel. The Kernel then processes the request and returns the result. Multiple processes may be simultaneously executing under the control of the Kernel. Whenever a system-dependent request (primarily a request for resource access) is desired, the Kernel is called to provide traffic control for communication between processes or with system resources.

The primary responsibility of a Kernel is the control of input devices, output devices, and secondary storage devices. If a process were capable of having direct access to these devices, there would be no way to force a process to respect the usage of a device by other processes. Thus, the Kernel is the only process capable of directly controlling these devices. If a process needs this device, it sends its request to the Kernel and the Kernel will do the processing. In this manner, the Kernel may restrict and supervise all critical operations.

A Kernel's code has traditionally been written in assembly language. This enhances the speed efficiency of the operating system. Assembly language generally allows a better method of control of the critical areas of a computer. Since assembly language is the direct representation of the machine language of the computer, anything the computer is capable of doing can be done in assembly language. When programming in assembly language, the execution speed of heavily used system routines can be optimized. If an operating system is being ported to a new computer processor, only the Kernel must be rewritten. This drastically improves the time necessary for conversion. This is a major reason for the success of the Unix operating system and other operating systems which incorporate this design idea.

2.3 The Solo operating system.

Brinch Hansen could see the power and flexibility of the Kernel design. He wanted a powerful operating system for development, but wanted it to be small enough for a single person to easily understand all the aspects of the operating system. One of his first attempts was the Solo operating system.[BH1] Solo provided an environment in which both sequential and concurrent Pascal programs could be easily developed. The Solo operating system was designed as a single user system and was small enough to be easily understood by a single person. This finally proved that an operating system did not have to be complex to be effective. By utilizing three primary processes to control input, execution, and output, Brinch Hansen provided a simple and effective usage of computing power. All three of these processes direct their critical tasks to a Kernel which controls all resources and resolves conflicting requests.

2.4 The Edison operating system.

Brinch Hansen was pleased with the results achieved by Solo but was still not satisfied with its size or the power available. Therefore, he proceeded to develop a new operating system to be utilized on a personal computer. The result was the Edison System. In this system, Brinch Hansen provides a powerful operating system with a closely related programming language. The entire system is written in the Edison programming language. The only exception is the Kernel which was written in the Alva assembly language. The Edison System continues his achievements in compact and portable operating systems. The system is still easily understood by a single user and is capable of powerful development software.

2.5 The Posix standardized operating system

The IEEE Standards Committee in January 1985 adopted the first draft of the specifications of a standardized operating system.[IEEE] The name of this standardized operating system is Posix. This standard (IEEE 1003.1) is based upon the Unix operating system. It provides for a standard operating system interface for all software developers to follow. Posix will be available upon many different manufacturer's computer architecture, but the software will be interchangeable from one system to another. It will not be executable-code compatible, but rather source-code compatible. The program will require compilation upon the new host system, but the program will not require any modification to execute correctly. This is a big step to establishing a common development environment for application software.

The IEEE Standards Committee is still working on a final draft for Posix standard 1003.1. The standard is also being reviewed by ANSI and is under consideration for adoption as an ISO standard. This will establish a world-wide standard in operating systems and software development environments.

The goals of the Posix standard and the goals of our project are the same. Both want to establish a standard environment which may be used to develop application software. The amount of development time will be greatly reduced since all software will only require implementation into one environment. Posix will require cooperation from all software and hardware vendors to succeed, the Edison System may be implemented by any interested person on most host environments. Posix will have greater execution speed because it is the operating system for the hardware, the Edison System can be implemented more quickly by layering the Edison System on top of the host operating system.

2.6 Improvement in portable operating systems.

The greatest hindrance to easily porting these later operating systems is the necessity of rewriting the Kernel in assembly language of the target machine. The Kernel does provide a centralized library of routines, but this library must be written specifically for each new computer environment. The intention of this project was to write a Kernel in a high-level language in order to improve its portability. No longer will it be necessary to rewrite the entire Kernel, but rather only small specific routines within the Kernel. The majority of the code consists of standard execution controls which would require no changes since the algorithms are not dependent upon system features. This benefit cannot be achieved without the cost of decreased speed efficiency. The use of an optimizing compiler can help to relieve this problem to some extent. The benefits greatly outweigh the costs so the change is a wise one.

To illustrate this concept, we have taken the Edison System and rewritten the assembly language Kernel into the high-level language C. We will adhere to the ANSI standard version of C. Variations of these standards are available in almost all the major computer environments. Therefore, the result will be a very portable Edison System which is capable of execution in a large variety of computer environments.

CHAPTER 3
THE EDISON SYSTEM

3.0 The Edison System

The Edison System contains three primary parts: the operating system, the Edison language, and utility programs. The combination of these three parts provides substantial power in establishing a good programming and application environment. The Edison language is based upon a strong relationship with the operating system. This is because the operating system is written in the Edison language. The features used to develop the operating system may also be used in developing application software. This strong similarity establishes an efficient relationship between the operating system and an application. The use of the exact same functions in both improves accuracy and aids in program development. The only portion of the operating system not written in Edison is the Kernel, which will be discussed later. The last portion of the Edison System consists of the assorted utilities which have been written to enhance the programming environment. The two most important of these utilities are the editor and the compiler. The editor is a simple line editor which provides for program creation and modification during development. The compiler is a four pass compiler which translates source code in the Edison language into a pseudo-code which is explained later in this chapter.

3.1 The Edison Operating System

The Edison Operating System is the result of the continuing work of Brinch Hansen. Beginning with his work on the Solo, Trio, and Mono operating systems and continuing with his work on Edison, it has been Brinch Hansen's goal to provide a powerful operating system that can be easily understood by the single programmer. The best way to provide this environment is through supplying only those features needed for applications. By reducing the total number of activities for the operating system to supervise, greater attention may be directed towards the supported routines. Fewer instructions also means less time is required during the testing and debugging stages of software development. The inclusion of more complex instructions would have created more time required to debug these instructions. It is much easier for a programmer to work with only a few simple instructions rather than to have complex instructions or many instructions available which the programmer cannot remember.

The major part of the Edison Operating System was written in the Edison language. This portion provides the functions the application programs will use. The processing of simpler functions is provided at this level. Whenever a process requires access to physical or logical devices, the operating system relays the request to the Kernel. The Kernel is the only portion of this operating system which is coded in assembly language. The Kernel processes all requests which are dependent upon the environment of the system.

The Kernel is the heart of the operating system. It is the portion that handles all critical and primary processing. Data exchange with the secondary storage, input and output transfers to devices, and process control are the concern of the Kernel. If an operating system is written correctly, the Kernel becomes the only portion of the operating system that will require modification in order to move to a new system environment.

When the computer system is initialized, the Kernel is the first section of code to be loaded. This will be done automatically by the computer from a predetermined device. The Kernel will initialize its environment (establish all default values and locations for this particular implementation) and then proceed to load the remainder of the operating system. At this point, the Edison environment is ready for the user's application.

The most important function of the Kernel in the Edison System is the processing of the pseudo-code generated by the compiler. Pseudo-code has been shown to provide increased portability between systems.[HEN] The pseudo-code instruction set for the Edison language is independent of the CPU instruction set of the executing machine. Thus, all compilers generate the same pseudo-code, regardless of the system environment in which the compiler exists.

The Kernel emulates a stack machine and provides for the translation from the pseudo-code instructions corresponding to a program into the actual machine instructions needed for execution. The use of pseudo-code does decrease the execution speed of a process when compared to the direct execution of machine instructions corresponding to a process. However, the increased flexibility of the pseudo-code instruction set makes the usage of pseudo-code a greatly desired requirement. The Edison System utilizes a pseudo-code instruction set with 67 operations.

The exchange of data information is the second most important task for the Kernel. The exchange of data may occur between main memory and secondary storage devices or I/O devices such as keyboards, crts, or printers. It is important during the usage of these devices that only one process should be accessing a particular device at any given instance. The supervision of data exchanges is the responsibility of the operating system. When an application process requests the use of a device from the operating system, the request is passed to the Kernel since it involves a system dependent feature. The Kernel will then restrict the execution of all of the other processes so that the device request may be completed before any other process can interrupt and possibly interfere with this request. The Kernel translates the application's request into a request directly to the device. This process works whether the device is a logical or physical device. If it is a logical device, the Kernel will perform any translations necessary to simulate the desired logical device.

The final task of the Kernel is management of memory allocation for variables and the correct treatment of data types. Memory allocation for variables forms a portion of the stack environment provided and used by the Kernel. If an implementation includes additional resources (memory or devices) it may be necessary to enlarge the size of the stack.

3.2 The Users View of Edison

The Edison System differs from traditional operating system command interpreters in the syntactic analysis which it performs upon user commands. Traditional command interpreters allow the parameters to be entered directly as a part of the command, for example:

```
print(systemtext, true)
```

This has proven to provide a very readable command structure, but is difficult for a command interpreter to process this type of command efficiently. The parameters may have to be in certain locations of the command string or proper interpretation may be dependent upon keywords in the command string. It is impossible for the command interpreter to know exactly how many parameters all application programs may require. Therefore, the interpreter cannot always catch errors in parameter passing. To avoid these problems, the Edison System uses a slightly different set of rules for command structure:

1. Only the application program name is accepted by the command interpreter.
2. It is the responsibility of each application program to request the necessary parameters during its execution.

The following example illustrates a typical user command in the Edison System:

```
Command = print

   File name = systemtext
   Print all pages? yes
   Print line numbers? yes
```

The message **Command =** is the prompt for the Edison System command interpreter. The application developer then defines prompts to request the parameters the application may require. These prompts are a part of the application software which must be coded by the programmer. The additional effect of this implementation is the increased user friendliness. The application software user is not required to remember the proper parameter sequence. The application program can be designed to request the information in a clear concise manner.

3.2 Size and Performance of the Edison System

The original version of the Edison System was developed by Brinch Hansen on a PDP 11. The final system for the original Edison consists of a total 10,000 lines of program text; the number of source lines in each of the major modules is shown in Table 3.1.

Operating System	1200 lines
Compiler	4200 lines
Editor	500 lines
Formatting program	400 lines
PDP 11 assembler	1600 lines
Other programs	400 lines
Edison programs	8300 lines
Kernel	1800 lines
Edison System	10100 lines

Source Code Size of the Edison System
Table 3.1

The main store for the system was contained in only 28K words and is illustrated in Table 3.2.

Kernel	1800 words
Operating system code	7200 words
Operating system variables	2400 words
System size	11400 words
User space	17300 words
Storage space	28700 words

Executable Code Size of the Edison System

Table 3.2

A simple benchmark speed test program was written to test the effects on execution speed. With the Kernel written in a high-level language, it could be predicted to execute slower than the original Edison System which utilizes an assembly language Kernel. The benchmark speed tests confirm this prediction. The (see Appendix C) executes on the original Edison System in 25 seconds. The same program executing in the portable Edison System requires 60 seconds to reach completion. While this execution speed difference is substantial, the benefits derived from a truly portable operating system are of greater importance to this project. The reduction in development time achieved by a common development environment is our primary goal.

3.3 Features of the Edison System

Several features of the Edison System make it a very user friendly environment. An important feature to users, is the consistency of editing features throughout the entire system. The command interpreter permits the use of the same editing controls which are employed by the editor. The editor is not a powerful, full-featured editor, but rather provides only the basic commands that are a requirement of a development editor. By reducing the number of commands available, it becomes easier for the user to master the editor in a very short time. The biggest drawback to the editor is the lack of full-screen editing. However, this feature requires system dependent functions for the control of the screen. Since this is counter-productive to the goal of the system, full-screen editing is not included.

The usability available to the user from Edison makes it a very good candidate as a language for writing application programs. The strong relationship between a high-level language and the operating system allows the opportunity for some very strong language features to be provided. The same routines which control the operating system, may also be used by the programmer to enhance the power of application program. (eg. Concurrent processes may be created and controlled by the user in an application program by using the same constructs the operating system uses when controlling application software.)

The programming language structure provides all the controls needed for the programmer to develop applications which utilize concurrent processing. Module structures are provided to contain routines to control critical areas. These modules are used by multiple processes to share common variables and devices. (eg. Only one process may send data to the printer, all other processes must wait until this process is finished with the printing task.)

The data types are well defined and provide ample control for the user to utilize. Strong type checking is inherent throughout the system. The data types available are illustrated in Table 3.3.

Integer	positive and negative whole numbers
Operators	arithmetic operators **+,-,*,div,mod**
Boolean	logical symbols denoting true or false
Operators	Boolean operators **not,and,or**
Characters	standard characters as defined by the ASCII character set
Enumerated	a collection of user defined symbols
Records	a collection of fields which may consist of any accepted data types
Arrays	a finite sequence of any accepted data type
Sets	a finite set of values, similar to the mathematical definition of sets

Data Types in the Edison Language

Table 3.3

The Edison System provides a powerful development and application environment. The original design allowed for easy porting of Edison to new computer system environments since all system dependent routines are contained in the Kernel. In order to move the Edison System to a new environment the Kernel is the only section of code which must be rewritten.

CHAPTER 4

SYSTEM IMPLEMENTATION

4.0 Procedures Used to Achieve the Project Goal

The primary goal for this project was to establish the feasibility of using a high level language to write a operating system Kernel. This should result in an operating system which may be easily ported to new system environments. Since this would then establish a standard operating environment, efficiency, in the development and maintenance of software applications can be greatly enhanced.

This first decision was related to which high level language to employ. It was necessary to use a language which provided powerful data structures, quick execution of resulting code, and compilers available for all major computer system environments. After a review of all major languages in use, the decision was made to utilize the language C. This choice was related to C's ability to fulfill the requirements:

> Powerful data structures - C provides for the standard data types, structures, and also includes enumerated types.

> Character strings are easily manipulated (which became a very powerful tool in the final implementation).

> Compilers are available for all major operating system environments, so there is a wide range of systems to which the Edison system can be ported.

Execution speed is a secondary concern of our project since its primary goal is to provide an implementation of an operating system which is extremely portable. It is recognized in the definition of the project that execution speed does suffer, but the reduction in speed is acceptable in most circumstances.

4.1 Conversion of Kernel code to C language

When Brinch Hansen originally developed the Edison system Kernel, he wrote the Kernel in the Edison language [BH2]. Later when he implemented the Edison system on a PDP 11, he converted the Edison code to the ALVA assembly language. The original Edison code was used by Brinch Hansen as comments to illustrate the program flow. We capitalize on the use of these comments. Part of the decision to use C was based upon the similarity of C and Edison languages. Approximately ninety percent of the Kernel is a simple conversion of Edison language syntax to the equivalent C language syntax. An illustration of this conversion between Edison and C may be seen with the routine 'writetext'.

```
proc writetext(value: text)
var i: int; c: char
begin i := 1;
     c := value[1];
     while c <> '#' do
          writechar(c);
          i := i + 1;
          c := value[i]
     end
end
```

Table 4.1 Edison code for procedure writetext

```
writetext(value)
text value;
{ int i;  char c;
     i := 1;
     c := value[1];
     while (c != '#')
        { writechar(c);
          i++;
          c := value[i]
        }
}
```

Table 4.2 C code for procedure writetext

4.1.1 Typical Problems Encountered in the Edison to C

Of course not everything was a straightforward conversion. The first problem to be encountered was a very minor one, but one which had disastrous effects. The memory allocation for a character in Edison is one word, which is two bytes long. In C, the memory allocation for a character is one byte long. The Edison system uses two bytes, but only the lower byte is the actual character. The upper byte is set to 0000 0000. The reason two bytes are used is because the stack is always accessed in terms of one word values. This is done in order to provide consistency throughout the system. The solution to this problem was simple. When the Kernel accessed the stack, it ignored the high byte and only used the lower byte when working with characters.

A similar problem also occurred with the memory allocation length of integers. Edison uses one word (two bytes) for the allocation of integers. However, C has two definitions of integers, a long integer occupies four bytes while a short integer has two bytes. In C, the integer data type **int** may be of either length. This is dependent upon the implementation of the compiler. It is important to check upon this feature whenever using a new C compiler for porting this Kernel. It is mandatory that the length of int be two bytes.

A much more difficult problem to overcome was the usage of sets. The Edison language provides the data type set. This definition is very similar to the mathematical definition of set. It is a collection of a finite number of members (possibly zero). The Edison definition requires that a set be of a basic type (char, integer, etc.) and that all members be of that type.

The C language does not include the data structure **set.** Upon investigation of the ALVA assembly language code, it was discovered that the implementation used there involved a bit-mapped field. A predetermined system limit of 128 values was chosen by Brinch Hansen as the maximum number of members in a set in the PDP 11 implementation. By using this same method, a structure was constructed in C which provided bit mapping. Each possible member (maximum 128) has a certain ordinal value. Whenever it is necessary to insert a member into a set, the ordinal value of the member is calculated and that bit is changed to the value of 1. Whenever it is necessary to remove a member of a set, the ordinal value of the member is again calculated and that bit is changed to the value of 0. The status of a member's inclusion may be determined at any time by checking its associated bit value..pa

4.2 The Use of Host Environment Calls

A problem results from installing Edison as an application layer upon a host operating system in that Edison does not have direct access to the disk storage devices. These operations fall under the control of the host operating system. For this reason, it is necessary for the Edison Kernel to make calls to the host environment to provide access to disk storage devices.

The C language provides the function **system**(string) where string denotes a character command string. C also includes several string manipulation functions which provide for the simple construction of lengthy command strings to be submitted to the host environment. Operations involving files such as directory lookups, file deletions, and file renaming can be provided by using the host environment's commands for these purposes. The Kernel functions create the needed command string sequence and submit it to the host system for processing. The results are then received by the Edison Kernel and relayed to the Edison operating system. (Application programs must request the Edison operating system to provide control for disk storage device operations.)

Routines involving access to secondary storage devices in the Kernel are dependant upon the host environment. These device routines must be changed for each new environment to which this Kernel is ported. These routines include: check_drive, copy_file, get_diskcatalog, rm_file, and protect_file. More information on porting to new host environments is given in the next chapter.

The remaining changes to the Kernel which involve input/output devices were accomplished without major changes to the methods previously used by Brinch Hansen. The C language provides for standard single character input/output routines. These routines (**getc()** and **printf("%c")**) are common with all C compilers and have presented no difficulties in their implementation. The single character routines provide the basic foundation for an implementation of all the Edison System's input/output routines, (eg. writename, writeline).

The conversion of a Kernel written in assembly language to a high-level language Kernel was accomplished with little difficulty. The interface to host environments was confined to five routines (check_drive, copy_file, get_diskcatalog, rm_file, and protect_file). This provided for straightforward conversions when porting to new hosts. By using a code-optimizing compiler, the reduction in execution speed was held to a minimum. A comparison was made between an original Edison System designed for an IBM PC compatible microcomputer and the MS-DOS implementation of our project. The benchmark program (see Appendix C) was executed in both environments. In the original Edison System, the execution was completed in 35 seconds. In the MS-DOS implementation, the same program required 60 seconds to execute. While this difference is substantial, it is acceptable under the goals defined by this project.

The layering of the Edison System above a host environment has provided an additional advantage. Now the Edison System is capable of utilizing greater disk storage than was possible with the original Kernel. Any disk storage that is available to the host environment is now available to the Edison System. Multiple disk drive access and expanded storage capacity will greatly enhance the usability of the Edison System.

CHAPTER 5

DESCRIPTION OF THE PROJECT ACTIVITY

5.0 Description of the Project Activity

The primary goal of this project was to provide a kernel which could be easily ported to new computer environments. In order to establish the effectiveness of the project, the kernel was ported to the following systems: UNIX System V, UNIX BSD release 4.3, Ultrix, and MS-DOS. It is not possible to transfer the executable code directly from one of these environments to another, therefore it is necessary to recompile the kernel from the C source code in each environment. All other components are written in Edison and compiled to virtual or pseudo code which the Kernel interprets.

5.1 Description of the Host Operating Systems

The host operating systems in the porting experiment are very similar to one another. They all represent a version of the original Unix operating system or a new operating system deriving many design concepts from Unix. These environments were not chosen in order to reduce the amount of code that must be changed, but rather because these operating systems were the only ones available with correctly functioning C compilers. Other operating systems did provide C compilers, but did not adhere to accepted industry standards in regard to available functions. It was a requirement that the C compiler be capable of all functions standard with both the ANSI C language and the language defined by Kernighan and Ritchie. Unfortunately, C compilers that meet this requirement are not available under all operating systems.

5.2 Routines Important to Porting

When rewriting the kernel, an effort was made to minimize the number of routines which would require modification during the porting process. There were seven routines which eventually required a large amount of modification to their logic. The routines contain the calls to the host operating system to provide the basic functions required by the Edison operating system. These functions provide input and output control to the console and storage devices. These functions are listed in Figure 5.1. The source code for these routines may be found in Appendix A. All other functions were capable of being rewritten in the standard C language.

1. check_drive()
2. copy_file()
3. get_diskcatalog()
4. rm_file()
5. protect_file()
6. readx()
7. writex()

Table 5.1 Functions Which Require Modification During Porting

5.2.1 check_drive()

The routine check_drive() is used to determine if the user of Edison has access to the requested drive. This is accomplished by requesting a listing of the files from the host environment. All host operating systems give an error response to this request when the drive is unavailable. It is necessary to modify the source code to include the request for the listing and the error message expected for the host environment.

5.2.2 copy_file()

This routine is requested whenever Edison wishes to copy an old file to a new file. This routine passes the file names to the host environment with the copy command of the host environment. Since this command syntax is not consistent among all operating systems, it is necessary to code a command specific to the target system into the routine when porting.

5.2.3 get_diskcatalog()

This routine processes the list of files available on the desired drive. This routine was the most difficult to code. It is necessary to code the command to request the list of files. This list must be directed into a file, which is then read from by this routine. It is necessary to modify the source code in this routine to include the locations of the file name, length of file in bytes, and access availability.

5.2.4 rm_file()

This routine deletes the desired file from the drive. It is necessary to code the host environment command for file deletion into this routine. This routine then adds the name of the desired file and submits the request to the host environment.

5.2.5 protect_file()

This routine sets the protection flag to the desired access status. When the protection flag is true, the file is read-only and cannot be altered or deleted. This routine requires the command syntax of the host environment command to alter file attributes. Some host environments (eg. MS-DOS) require a resident command file to be present (eg. ATTRIB.EXE for MS-DOS).

5.2.6 readx() and writex()

These routines read and write a character to the host environment display. Each of these routines utilizes the standard input/output routines of the C language. An enhancement sometimes available to the C language in host environments is a library called curses. If the curses enhancements are available, then different input/output functions are used in the implementation. Curses is used because it provides full screen control in the host environment. It is necessary to use different function calls when utilizing this library. For this reason, it is necessary to tell these routines which routines to use..pa

5.3 Techniques Used During This Project

During this project it was desired to provide a simple method to allow the kernel to be ported from one of the test host environments to another. The goal was to have one source code file for the kernel which required flags to be set at the beginning of the code. In this manner, it was easier to port the file from one environment to another and also to better illustrate the routines which require modification during porting. By setting the flags as shown in Figure 2, it is possible to compile the kernel for the desired host environment.

Host Environment	Flag Settings
MS-DOS	#define MSDOS 1
UNIX System V	#define UNIXSYSTEM 1
UNIX BSD4.3	#define UNIXSYSTEM 0
Ultrix	#define UNIXSYSTEM 0
HCX/UX 3.0	#define UNIXSYSTEM 0

Flag Settings for Test Environments
Table 5.2

Because of desired file naming conventions, the Edison system is invoked from any host environment system by the command 'Edison'. The source code for this file reveals that this is a simple routine which calls the kernel. The kernel initializes the Edison system and processes the command instructions until controlled shutdown or an abnormal ending.

To increase the flexibility of the Edison system, a default drive was encoded to allow for two data drives to be accessed and still have the system utilities available to the user. This drive and directory are defined in the kernel as DEFAULTDRIVE. It is necessary to provide this information to the kernel before compilation.

5.4 Summary

This primary goal of this project was to provide a kernel which could be easily ported to new computer environments. In this manner, a uniform environment would be available to all Edison programs in all computer environments. This project achieved its goal by writing the kernel in the high-level language C. Only seven routines require modification to their logic in order to port the Edison system to a new host environment.

REFERENCES

[BH1] Hansen, Per Brinch. The Architecture of Concurrent Programs, Prentice-Hall, 1977.

[BH2] Hansen, Per Brinch. Programming a Personal Computer, Prentice-Hall, 1982.

[HEN] Henderson, John. Software Portability, Gower Technical Press Limited, 1988.

[IBM] IBM Systems Journal, Volume 18, Number 1, 1979, p. 1.

[IEEE] IEEE Software, Posix Success Enhances CS's Standards Leadership, January 1989, p. 108,131.

[K&R] Kernighan, Brian W. and Ritchie, Dennis M.. The C Programming Language, Prentice-Hall, Englewood Cliffs, 1978.

[PET] Peterson, James and Silberschatz, Abraham. Operating System Concepts, Addison-Wesley Publishing, 1983.

[SCO] Scott, Terry. An Implementation of the KERMIT Protocol Using the Edison System, Kansas State University, 1985.

[WIL] Wilde, Martin. Solo32: A Concurrent Pascal Operating System with Unix Interfaces, Kansas State University, 1984.

APPENDIX A

KERNEL FUNCTIONS REQUIRING MODIFICATION

```
/************************************************************
This is a new procedure to be passed to the operating system
as a procedure parameter.  It is used to determine if a
directory input by the user is a valid directory.  The
parameter 'value' in the following sample call is returned
TRUE if the directory exists and the user has access to it,
and FALSE otherwise.

proc k_checkdrive(drivedir: overline; drive: int;
                  var value: bool);
parameters           at          procedure           entry:
st[s -  0] : value      *  address of bool indicating if   *
                       *  drivedir is a valid directory   *
st[s -  1] : drive      *  (int)    drive number (1 or 2)  *
st[s  -  81]  :  drivedir   *   array  [1:81]  (char)
************************************************************/
check_drive()
{ integer x=0, driveno;
char c, *fname, line[100];
FILE *fp, *fopen();

#if TRACE >= 1
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In check_drive P=%ld s=%ld t=%ld\n",p, s,
          t);
fclose(fp);
#endif
s -= 82;
c = st[s + 1];
driveno = st[s + 81]; /* get the drive number */

  /* Copy the directory to be checked into a string */
while ((x <= 80) && (    ((c >= 'a') && (c <= 'z'))
   || ((c >= '-') && (c <= '9'))
   || ((c >= 'A') && (c <= 'Z'))
   ||   c == '/'  ||  c == '\\' || c == ':'))
   { drive[driveno][x++] = c;
   c = st[s + 1 + x];
   }
#ifdef UNIXSYSTEM
if (drive[driveno][x-1] != '/')
   drive[driveno][x++] = '/';
#endif
#ifdef MSDOS
if (drive[driveno][x-1] != '\\')
   drive[driveno][x++] = '\\';
#endif

drive[driveno][x] = '\0';
```

```
/* Get a temporary file and direct a directory listing    */
/* for the directory to be checked into the temporary file*/
gettempfile(&fname);
#ifdef MSDOS
sprintf(systemcall, "dir %s > %s", &drive[driveno][0],
fname);
#else
sprintf(systemcall,   "ls   -l   %s   >   %s   2>&1",
&drive[driveno][0], fname);
#endif
#if TRACE < 0
printf("%s - Called from check_drive()\n",systemcall);
#endif
system(systemcall);

  /* Open the temporary file and read the first line */
fp = fopen(fname, "r");
#ifdef MSDOS
fscanf(fp, "%[^\n]s", line);
fscanf(fp, "%[^\n]s", line);
fscanf(fp, "%[^\n]s", line);
fscanf(fp, "%[^\n]s", line);
fscanf(fp, "%[^\n]s", line);
if (feof(fp))
   st[st[s + 82]] = FALSE;
else
   st[st[s + 82]] = TRUE;
#else
fscanf(fp, "%[^\n]s", line);

/* If the directory exists and the user has access, the  */
/* first line will have "total " for the first six
   characters */
if (strncmp(line, "total ", 6))
   st[st[s + 82]] = FALSE;
else
   st[st[s + 82]] = TRUE;
#endif
  /* close and remove the temporary file */
fclose(fp);
#ifdef MSDOS
sprintf(systemcall, "erase %s > NUL", fname);
#else
sprintf(systemcall, "rm %s", fname);
#endif
#if TRACE < 0
printf("%s - Called from check_drive()\n",systemcall);
#endif
system(systemcall);
```

```
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

/**********************************************************
 This is a new procedure to be passed to the operating
system as a procedure parameter.  It is used to copy files.


proc k_copy_file(file1, file2: overname; drive1,
                 drive2: int);

parameters           at           procedure         entry:
st[s     -       0]    :    drive2                   (int)
st[s     -       1]    :    drive1                   (int)
st[s   -  13]   :  file2           array  [1:12]  (char)
st[s - 25] : file1      array [1:12] (char)

**********************************************************/
copy_file()
{ char file1[13], file2[13];

#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In copy_file P=%ld s=%ld t=%ld\n",p, s, t);
fclose(fp);
#endif
s -= 26;

  /* if the first file is "*" then copy all */
  /* files from directory 1 to directory 2  */
if (st[s + 1] == '*')
#ifdef MSDOS
   sprintf(systemcall, "copy %s* %s > NUL", &drive[st[s +
25]][0],
   &drive[st[s + 26]][0]);
#else
sprintf(systemcall, "cp %s* %s", &drive[st[s + 25]][0],
&drive[st[s + 26]][0]);
#endif
else
   { getfilename(file1, s + 1);
   getfilename(file2, s + 13);
#ifdef MSDOS
```

```
   sprintf(systemcall, "copy %s%s %s%s > NUL", &drive[st[s +
25]][0], file1,
   &drive[st[s + 26]][0], file2);
#else
   sprintf(systemcall, "cp %s%s %s%s", &drive[st[s +
25]][0], file1,
   &drive[st[s + 26]][0], file2);
#endif
   }
#if TRACE < 0
printf("%s - Called from copy_file()\n",systemcall);
#endif
system(systemcall);
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

/************************************************************
This is a new procedure to be passed to the operating
system as a procedure parameter.  It is used to get the
directory of a directory which is used as a virtual foppy
disk.

proc k_get_diskcatalog(var catalog: overcatalog;
                              drive: int);
parameters at procedure entry:

st[s   -      0]   :   drive                  (int)
st[s -  1] : catalog   address of catalog variable

The       catalog       has       the       form:
size:                                        int;
contents:                                  table;
unused: filler;

table          has          the          form:
array [1:47] (item)

item          has          the          form:
title:                                      name;
attr: attributes

attributes          has          the          form:
address:                                     int;
length:                                 position;
protected:                                  bool;
```

```
position            has            the            form:
pages,                  words:                     int;

So from a base address of base, the overall structure on the
stack                     looks                       like:
st[base +  0] : size     "number of item entries in catalog"

item1base          =          base          +          1
st[item1base      +              0]       :        title
st[item1base      +      12]       :        attr.address
      "address is not needed without real disks to map"
st[item1base + 13] : attr.position.pages "# of pages"
st[item1base + 14] : attr.position.words
                              "# words on last page"
st[item1base     +      15]      :      attr.protected
item2base        =        item1base        +        16
st[item2base      +              0]       :        title
st[item2base      +      12]       :        attr.address
          "address is not needed without real disks to map"
st[item2base + 13] : attr.position.pages "# of pages"
st[item2base + 14] : attr.position.words
                              "# words on last page"
st[item2base     +      15]      :      attr.protected
item47base        =        item46base        +        16
*************************************************************/
#ifdef MSDOS                    /* MS-DOS version  of
get_diskcatalog */
get_diskcatalog()
{ long itemaddr, value, count=0;
FILE *fp, *fopen();
char *fname, line[100], *ptr, *login;
integer w;

#if TRACE >= 1
fp = fopen("log","a");
if (printit) fprintf(fp, "In get_diskcatalog P=%ld s=%ld
          t=%ld\n",p, s, t);
fclose(fp);
#endif
s -= 2;
itemaddr = st[s + 1] + 1;
gettempfile(&fname);
w = strlen(drive[st[s+2]][0]);
strncat(strcpy(systemcall, "dir "), &drive[st[s + 2]][0],w);
strcat(strcat(systemcall, " > "), fname);
#if TRACE < 0
printf("%s - Called from get_diskcatalog()\n",systemcall);
#endif
system(systemcall);
```

```
fp = fopen(fname, "r");
   for (w=1; w<5; w++) {          /*  Remove the first 4
                                  lines of the directory */
   fscanf(fp,"%[^\n]s",line);
   fgetc(fp);
   }
while (!(feof(fp)))
   { fscanf(fp,"%[^\n]s",line);
   fgetc(fp);
   if ((count < 47) &&  (strncmp(&line[10],"File(s)",7)) &&
      (strncmp(&line[13],"<DIR>",5)) &&
      (!(feof(fp))) && (strcmp(fname, &line[0])))
      {
      for (w=0; line[w]!='\0'; w++)
          /* Make all files lower case */
         line[w] = tolower(line[w]);
      st[itemaddr + 15] = FALSE;
      ptr = &line[13];
      value = 0;
      while ((*ptr < '0') || (*ptr > '9'))
         ptr++;
      while ((*ptr >= '0') && (*ptr <= '9'))
         { value = value * 10 + (*ptr - '0');
         ptr++;
         }
      value = value / 2;
      st[itemaddr + 13] = value / PAGESIZE + 1;
      if (!(st[itemaddr + 14] = value % PAGESIZE))
         { st[s + 13] -= 1;
         st[s + 14] = PAGESIZE;
         }
      value = 0;
      while ((value < 12) && (line[0 + value] != '\0'))
         { st[itemaddr + value] = line[0 + value];
         value++;
         }
      while (value < 12)
         st[itemaddr + value++] = ' ';
      count++;
      itemaddr += 16;
      }
   }
st[st[s + 1]] = count;
fclose(fp);
strcat(strcpy(systemcall, "erase "), fname);
#if TRACE < 0
printf("%s - Called from get_diskcatalog()\n",systemcall);
#endif
system(systemcall);
#if TRACE >= 2
```

```
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}
#endif

#ifdef UNIXSYSTEM

/* UNIXSYSTEM Version of get_diskcatalog() */

get_diskcatalog()
{ integer itemaddr, value, count=0;
FILE *fp, *fopen();
char *fname, line[100], *ptr, *login;

#if TRACE >= 1
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In get_diskcatalog P=%ld s=%ld t=%ld\n",p,
          s, t);
fclose(fp);
#endif
s -= 2;

  /* find the name of the user */
login = getpwuid(getuid())->pw_name;

  /* set itemaddr to point to the first item in catalog */
itemaddr = st[s + 1] + 1;

/* Get a temporary file and direct a directory listing */
/* for the directory to be checked into the temporary file*/
gettempfile(&fname);
sprintf(systemcall, "ls -l %s > %s", &drive[st[s + 2]][0],
fname);
#if TRACE < 0
printf("%s - Called from get_diskcatalog()\n",systemcall);
#endif
system(systemcall);

  /* Open the temporary file and skip over the first line */
fp = fopen(fname, "r");
fscanf(fp,"%[^\n]s",line);
fgetc(fp);

  /* read in the first line of the directory listing */
fscanf(fp,"%[^\n]s",line);
fgetc(fp);
```

```
/*************************************************************
for each line in the directory listing put the
file in the catalog if it meets certain criteria
criteria                                                   :
catalog only holds 47 files. Each line must represent a
file.   "line[0] == '-' represents a file. The filename must
be less or equal to 12 characters in length.  Don't include
the kernel's temporary file.  Files are ignored if the user
does not have read access.
*************************************************************/
while (!(feof(fp)))
   { if ((count < 47)              &&
      (line[0] == '-')             &&
      (strlen(&line[45]) <= 12)    &&
      (strcmp(fname, &line[45]))  &&
      ((strncmp(&line[14], login, strlen(login))) ?
      (line[7] == 'r') : (line[1] == 'r')))
      {
      if (strncmp(&line[14], login, strlen(login)) == 0)
         st[itemaddr + 15] = (line[2] == '-');
      else
         st[itemaddr + 15] = (line[8] == '-');

      /* read the size of the file in bytes */
      ptr = &line[20];
      value = 0;
      while ((*ptr < '0') || (*ptr > '9'))
         ptr++;
      while ((*ptr >= '0') && (*ptr <= '9'))
         { value = value * 10 + (*ptr - '0');
         ptr++;
         }

      /* convert from bytes to words (two byte words) */
      value = value / 2;

      /* determine number of pages and number of      */
      /* words on last page from the number of words */
      st[itemaddr + 13] = value / PAGESIZE + 1;
      if (!(st[itemaddr + 14] = value % PAGESIZE))
         { st[s + 13] -= 1;
         st[s + 14] = PAGESIZE;
         }

      /* copy the filename */
      value = 0;
      while ((value < 12) && (line[45 + value] != '\0'))
         { st[itemaddr + value] = line[45 + value];
         value++;
```

```
           }

        /* fill the unused part of the name with blanks */
        while (value < 12)
           st[itemaddr + value++] = ' ';

        /* increment the count and point to the next item */
        count++;
        itemaddr += 16;
        }

      /* read the next line */
     fscanf(fp,"%[^\n]s",line);
     fgetc(fp);
     }

   /* put the number of items in the catalog */
 st[st[s + 1]] = count;

   /* close and remove the temporary file */
 fclose(fp);
 sprintf(systemcall, "rm %s", fname);
 #if TRACE < 0
 printf("%s - Called from get_diskcatalog()\n",systemcall);
 #endif
 system(systemcall);
 #if TRACE >= 2
 register_dump();
 #endif
 #if TRACE >= 3
 stack_dump(3);
 #endif
 }
 #endif      /* UNIXSYSTEM Version */

/************************************************************
 This is a new procedure to be passed to the operating
system as a procedure parameter.  It is used to delete
files.  If the file to be removed is 'halt.command' and the
directory  number is number 5 (an invalid drive), then the
kernel  exits gracefully (ie the user has typed in the
command 'exit').


proc  k_rm_file(file1:  overname;  drive1:  int);
parameters at procedure entry:
   st[s   -     0]   :  drive1          (int)
st[s - 12] : file1     array [1:12] (char)
*************************************************************/
rm_file()
```

```
{ char file[13];

#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In rm_file P=%ld s=%ld t=%ld \n", p, s, t);
fclose(fp);
#endif
s -= 13;
getfilename(file, s + 1);
if ((strcmp(file, "halt.command") == 0) && (st[s+13] == 5))
   quit(0);
#ifdef MSDOS
sprintf(systemcall, "erase %s%s > NUL", &drive[st[s+13]][0],
file);
#endif
#ifdef UNIXSYSTEM
sprintf(systemcall, "rm %s%s", &drive[st[s+13]][0], file);
#endif
#if TRACE < 0
printf("%s - Called from rm_file()\n",systemcall);
#endif
system(systemcall);
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

/*********************************************************
This is a new procedure to be passed to the operating system
as a procedure parameter. It is used to protect and unpro-
tect files.

proc k_protect_file(title: overname; drive: int;
                    value: bool))
parameters         at         procedure         entry:
st[s -  0] : value     (bool)
TRUE:protect  FALSE:unprotect
st[s   -     1]   :   drive                      (int)
st[s - 13] : title     array [1:12] (char)
*********************************************************/
protect_file()
{ char file[13];

#if TRACE >= 1
FILE *fp, *fopen();
```

```
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In protect_file P=%ld s=%ld t=%ld\n",p, s,
         t);
fclose(fp);
#endif
s -= 14;
getfilename(file, s + 1);
#ifdef MSDOS   /* Requires MS-DOS version 3.x  ATTRIB.EXE */
sprintf(systemcall, "attrib %s %s%s", (st[s+ 14] ? "+R" :
"-R"),
&drive[st[s+13]][0], file);
#endif
#ifdef UNIXSYSTEM
sprintf(systemcall, "chmod %s %s%s", (st[s + 14] ? "ugo-w" :
"u+w"),
&drive[st[s+13]][0], file);
#endif
#if TRACE < 0
printf("%s - Called from protect_file()\n",systemcall);
#endif
system(systemcall);
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

/************************************************************/
/* The procedure readx has been rewritten.                  */
/* It no long works directly with the keyboard.  Instead */
/* it simply calls routines to get the key value.          */
/************************************************************/
readx(value)
char *value;{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In readx P=%ld s=%ld t=%ld \n",p, s, t);
fclose(fp);
#endif
```

```
#ifdef UNIXSYSTEM
*value = getch();
#else
*value = getchar();
#endif                              /* Translate Unix new-
line to */
if (*value==10) *value=13;     /* Edison newline */
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}

/*****************************************************/
/* The procedure write has been rewritten.           */
/* It no long works directly with the screen memory. */
/* Instead it simply calls routines to place the     */
/* character value.                                  */
/* It has also been renamed to writex.               */
/*****************************************************/
writex(value)
integer value;
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In writex P=%ld s=%ld t=%ld %c (%ld)\n", p,
          s, t, value,value);
fclose(fp);
#endif




#ifdef UNIXSYSTEM
addch(value);
refresh();
#else
printf("%c",value);
#endif
#if TRACE >= 2
```

```
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}
```

APPENDIX B

THE EDISON SYSTEM KERNEL

```
#define MSDOS 1

#include <stdio.h>

#ifdef UNIXSYSTEM
#include <pwd.h>
#include <signal.h>
#include <curses.h>
#endif

typedef long integer;  /* Define length of integer for
compiler */

#define OPSYSCODE 12514   /* Size of Operating System */

#define DEFAULTDRIVE "c:\\edison\\"
/*
DEFAULTDRIVE is the directory that is checked last for
the programs the user tries to execute.  It is the 'third'
floppy disk drive.
*/

#define TRACE 0
/*      TRACE level -1   -    Host system calls
        TRACE level  0   -    No trace statements
        TRACE level  1   -    Procedure entry/exit
        TRACE level  2   -    Register dump
        TRACE level  3   -    Stack dump

All but one opcode procedures and many other procedures have
the trace statements.  Newline is the only opcode procedure
without the trace statements because it has been removed
from the output of the compiler as an optimization.  All
trace and debugging statements are printed to a file named
log in order to not interfer with running the system.
*/


#define TRUE 1
#define FALSE 0
#define FILLERWORD 0
#define PAGESIZE 512
#define SECTORLENGTH 64
#define MINADDR 999

/* 20 communication channes have been included
   for 'distributed' programs. */

#define NUMCOMMCHANNELS   20
```

```
#define MAXADDR             57342

#ifdef MSDOS
typedef long huge store[MAXADDR + NUMCOMMCHANNELS + 1];
#else
typedef  int store[MAXADDR + NUMCOMMCHANNELS + 1];
#endif

/* Here are the various opcodes that are to */
/* be executed, and their respective value  */
#define ADD4          1792
#define ALSO4         1795
#define AND4          1798
#define ASSIGN4       1801
#define BLANK4        1804
#define COBEGIN4      1807
#define CONSTANT4     1810
#define CONSTRUCT4    1813
#define DIFFERENCE4   1816
#define DIVIDE4       1819
#define DO4           1822
#define ELSE4         1825
#define ENDCODE4      1828
#define ENDLIB4       1831
#define ENDPROC4      1834
#define ENDWHEN4      1837
#define EQUAL4        1840
#define FIELD4        1843
#define GOTO4         1846
#define GREATER4      1849
#define IN4           1852
#define INDEX4        1855
#define INSTANCE4     1858
#define INTERSECTION4 1861
#define LESS4         1864
#define LIBPROC4      1867
#define MINUS4        1870
#define MODULO4       1873
#define MULTIPLY4     1876
#define NEWLINE4      1879
#define NOT4          1882
#define NOTEQUAL4     1885
#define NOTGREATER4   1888
#define NOTLESS4      1891
#define OR4           1894
#define PARAMARG4     1897
#define PARAMCALL4    1900
#define PROCARG4      1903
#define PROCCALL4     1906
```

```
#define PROCEDURE4     1909
#define PROCESS4       1912
#define SUBTRACT4      1915
#define UNION4         1918
#define VALSPACE4      1921
#define VALUE4         1924
#define VARIABLE4      1927
#define WAIT4          1930
#define WHEN4          1933
#define ADDR4          1936
#define HALT4          1939
#define OBTAIN4        1942
#define PLACE4         1945
#define SENSE4         1948
#define ELEMASSIGN4    1951
#define ELEMVALUE4     1954
#define LOCALCASE4     1957
#define LOCALSET4      1960
#define LOCALVALUE4    1963
#define LOCALVAR4      1966
#define OUTERCALL4     1969
#define OUTERCASE4     1972
#define OUTERPARAM4    1975
#define OUTERSET4      1978
#define OUTERVALUE4    1981
#define OUTERVAR4      1984
#define SETCONST4      1987
#define SINGLETON4     1990
#define STRINGCONST4   1993


store st;

#define NONE 0
#define NL 10
#define SP ' '
#define ESC 27
#define pdp11 FALSE
#define MAXROW 24
#define MAXCOLUMN 80
#define TEXTLENGTH 80
#define NAMELENGTH 12
#define SETLENGTH 8
#define SETLIMIT 127
#define CURSORNO 0
#define ERASENO 1
#define DISPLAYNO 2
#define ACCEPTNO 3
#define PRINTNO 4
#define GETNO 5
```

```
#define PUTNO 6

#define CHECKDRIVENO 7
#define COPYNO 8
#define GETCATALOGNO 9
#define RMNO 10
#define PROTECTNO 11
#define PARAMLENGTH 27
#define MAXPARAM 12

/* #define PARAMLENGTH 17
   #define MAXPARAM 7 */
#define MAXPROC 6


typedef char text[TEXTLENGTH + 1];
typedef char name[NAMELENGTH + 1];
typedef struct
{ integer settype2[8];
} settype1;

struct procstate
{ integer bx, sx, tx, px;
};
struct procstate q[MAXPROC + 1];

integer linenumber;


integer this, tasks, stacktop, progtop, b, s, t, p, printit;

char drive[3][81], systemcall[193];
/* drive is an array of directories that are  */
/* being used as virtual floppy disks         */
/* drive[0] and drive[1] are user directories */
/* drive[2] is the default drive              */

loadset(addr, value)
integer addr;
settype1 *value;
{ integer i=0;
while (i<SETLENGTH)
   {
   value->settype2[i] =
   st[addr + i];
   i++;
   }
}

storeset(addr, value)
```

```
integer addr;
settype1 *value;
{ integer i=0;
while (i<SETLENGTH)
   {
   st[addr + i] =
   value->settype2[i];
   i++;
   }
}

loadname(addr, value)
integer addr; name value;
{ integer i = 0;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In loadname P=%ld s=%ld t=%ld %ld '%s'\n",p,
          s, t, addr, value);
fclose(fp);
#endif

while (i < NAMELENGTH - 1)
   { value[i] = st[addr + i];
   i++;
   }
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}



findname(id)
name id;
{ integer addr;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In findname P=%ld s=%ld t=%ld '%s'\n", p, s,
          t, id);
fclose(fp);
#endif

if (tasks == 1)
```

```
   addr = t + 1;
else
   addr = progtop + 1;
while (addr < p)
   { loadname(addr, id);
   addr += NAMELENGTH;

   addr += (st[addr] / 2);
   }

#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}



/*****************************************************/
/* The procedure write has been rewritten.          */
/* It no long works directly with the screen memory. */
/* Instead it simply calls routines to place the     */
/* character value.                                  */
/* It has also been renamed to writex.               */
/*****************************************************/
writex(value)
integer value;
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In writex P=%ld s=%ld t=%ld %c (%ld)\n", p,
          s, t, value,value);
fclose(fp);
#endif





#ifdef UNIXSYSTEM
addch(value);
refresh();
#else
printf("%c",value);
#endif
```

```
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}


/**********************************************************/
/* The procedure write has been modified.                 */
/**********************************************************/
writechar(value)
integer value;{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In writechar P=%ld s=%ld t=%ld %c (%ld)\n",
          p, s, t,value,value);
fclose(fp);
#endif




writex(value);

if (value == NL) writex(13);
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

writetext(value)
text value;
{ integer i = 0; char c;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In writetext P=%ld s=%ld t=%ld '%s'\n", p,
          s, t,value);
fclose(fp);
#endif
```

```
c = value[0];
while (c != '#')
   { writechar(c);
   c = value[++i];
   }
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}

writeint(value)
integer value;
{ char no[7];
integer i;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In writeint P=%ld s=%ld t=%ld %ld\n", p, s,
          t,value);
fclose(fp);
#endif

if (value == 0)
   { i = 1; no[1] = '0';}
else
   if (value > 0)
      { i = 0;
      while (value > 0)
         { no[++i] = value % 10
         + '0';
         value = value / 10;
         }
      }
while (i > 0)
   writechar(no[i--]);


#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}

writename(value)
```

```
char value[];
{ integer i = 0; char c;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In writename P=%ld s=%ld t=%ld '%s'\n", p,
          s, t,value);
fclose(fp);
#endif

while (i < NAMELENGTH - 1)
   { c = value[i++];
   if (c != SP)
      writechar(c);
   }


#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}

/**********************************************************/
/* The procedure readx has been rewritten.                */
/* It no long works directly with the keyboard.  Instead */
/* it simply calls routines to get the key value.         */
/**********************************************************/
readx(value)
char *value;{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In readx P=%ld s=%ld t=%ld \n",p, s, t);
fclose(fp);
#endif
```

```
#ifdef UNIXSYSTEM
*value = getch();
#else
*value = getchar();
#endif                              /* Translate Unix new-
line to */
if (*value==10) *value=13;      /* Edison newline */
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}

pausex()
{ char response;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In pausex P=%ld s=%ld t=%ld\n",p, s, t);
fclose(fp);
#endif
writetext(
"push return to continue#");


readx(&response);
writechar(NL);
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}

stop(lineno, reason)
integer lineno; text reason;
{ name id;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In stop  P=%ld s=%ld t=%ld %ld '%s'\n",p, s,
           t, lineno, reason);
fclose(fp);
#endif
```

```
findname(id);
writechar(NL);
writename(id);
writetext(" line #");

writeint(lineno);
writechar(SP);
writetext(reason);
writechar(NL);

reboot();
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(30);
#endif
}

/* The procedures processor_trap, instruction_trap */
/* and power trap are no longer needed.            */

processlimit(lineno)
integer lineno;{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In processlimit  P=%ld s=%ld t=%ld %ld\n",p,
          s, t,lineno);
fclose(fp);
#endif
stop(lineno,
"process limit exceeded#");

#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}

variablelimit(lineno)
integer lineno;{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In variablelimit  P=%ld s=%ld t=%ld
          %ld\n",p, s, t,lineno);
fclose(fp);
#endif
stop(lineno,
"variable limit exceeded#");

#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}

rangeerror(lineno)
integer lineno;{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In rangeerror  P=%ld s=%ld t=%ld %ld\n",p,
          s, t, lineno);
fclose(fp);
#endif
stop(lineno,
"range limit exceeded#");
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}

callerror(lineno){
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In callerror  P=%ld s=%ld t=%ld %ld\n",p, s,
          t, lineno);
fclose(fp);
#endif
stop(lineno,
"invalid program call#");
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
```

```
#endif
}

#define MAXCODE 12288
moveprogram()
{ integer m;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In moveprogram P=%ld s=%ld t=%ld \n", p, s,
          t);
fclose(fp);
#endif

m = (st[s - MAXCODE + 1] / 2) +
NAMELENGTH;
s = s - MAXCODE - NAMELENGTH;

t -= m;
while (m > 0)
   { st[t + m] = st[s + m];

   m--;
   }

#if TRACE >= 1
   {FILE *fp, *fopen();
   integer x=1;
   fp = fopen("log", "a");

   fprintf(fp, " moving code for program : ");
   while (x <= NAMELENGTH)
      fprintf(fp, "%c", st[t + x++]);
   fprintf(fp, " starting at %ld  (t+1)\n", t+1);
   fclose(fp);
   }
#endif

#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}


/**************************************************************
The new version of loadprogram sets up the stack in order to
```

```
to call the procedure get.  get has already been modified to
read a page from a Unix file.  So loadprogram only needs to
call get multiple times and read in the entire file.
************************************************************/
loadprogram()
{ integer i;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In loadprogram P=%ld s=%ld \n",p, s);
fclose(fp);
#endif
st[s + 1] = 'o';
st[s + 2] = 'p';
st[s + 3] = 's';
st[s + 4] = 'y';
st[s + 5] = 's';
st[s + 6] = 'c';
st[s + 7] = 'o';
st[s + 8] = 'd';
st[s + 9] = '.';
st[s + 10] = 'p';
st[s + 11] = 'l';
st[s + 12] = 'n';

st[s + 13] = 2;         /* Drive number */
st[s + 14] = 1;         /* Page number */

i = OPSYSCODE / 2;
st[s + 15] = i / PAGESIZE + 1;    /* st[s + 15] = number of
pages */
if (!(st[s + 16] = i % PAGESIZE)) /* st[s + 16] = number of
words */
   { st[s + 15] -= 1;             /*                    on
the last page */
   st[s + 16] = PAGESIZE;
   }
st[s + 17] = s + 18 ;  /* Address to load the page */
while (st[s + 14] <= st[s + 15])
   { s += 17;
   get();
   st[s + 14] = st[s + 14] + 1;
   st[s + 17] = st[s + 17] + PAGESIZE;
   }
i = NAMELENGTH + 1;
while (--i > 0)
   st[s + 5 + i] = st[s + i];
s += MAXCODE + 17; /* patch */
moveprogram();
s = s - 17 + NAMELENGTH;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}


resume(){
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In resume P=%ld s=%ld t=%ld\n",p, s, t);
fclose(fp);
#endif
b = q[this].bx;
s = q[this].sx;
t = q[this].tx;
p = q[this].px;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


preempt(){
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In preempt P=%ld s=%ld t=%ld\n",p, s, t);
fclose(fp);
#endif
q[this].bx = b;
q[this].sx = s;
q[this].tx = t;
q[this].px = p;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}
```

```
/*****************************************************/
/* This procedure has been slightly modified.        */
/* New kernel calls have to be defined as parameters */
/* to the operating system.  Also the default drive  */
/* is initialized.                                   */
/*****************************************************/
initialize(){
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In initialize P=%ld s=%ld t=%ld\n",p, s, t);
fclose(fp);
#endif


#ifdef UNIXSYSTEM
initsignals();
initcurses();
#endif

this = 1;
tasks = 1;
b = MINADDR + PARAMLENGTH;

st[b - 27] = pdp11;

st[b - 26] = MAXROW;
st[b - 25] = MAXCOLUMN;
st[b - 23] = CURSORNO;
st[b - 21] = ERASENO;
st[b - 19] = DISPLAYNO;
st[b - 17] = ACCEPTNO;
st[b - 15] = PRINTNO;
st[b - 13] = GETNO;
st[b - 11] = PUTNO;
st[b - 9] = CHECKDRIVENO;
st[b - 7] = COPYNO;
st[b - 5] = GETCATALOGNO;
st[b - 3] = RMNO;
st[b - 1] = PROTECTNO;
strcpy(&drive[2][0], DEFAULTDRIVE);
s = b + 4;
st[s] = NONE;

t = MAXADDR;
loadprogram();
p = t + NAMELENGTH + 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}


/*****************************************************/
/* This procedure has been rewritten.                */
/* If curses is being used the cursor is moved to    */
/*     correct screen coordinates.                   */
/* else a message is printed saying this function is */
/*     is no longer operative.                       */
/*****************************************************/
cursor()
{ integer row, column;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In cursor P=%ld s=%ld t=%ld\n",p, s, t);
fclose(fp);
#endif

s -= 2;
row = st[s + 1];
column = st[s + 2];


#ifdef UNIXSYSTEM
move(row - 1, column - 1);
refresh();
#else
printf("\ncursor is now an inoperative function\n");
#endif
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

/*****************************************************/
/* This procedure has been rewritten.                */
/* If curses is being used the function is performed */
/* else a message is printed saying this function is */
/*     is no longer operative.                       */
/* It has also been renamed to kerase.                    */
/*********************************************************/

kerase(){
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In kerase P=%ld s=%ld t=%ld\n",p, s, t);
fclose(fp);
#endif


#ifdef UNIXSYSTEM
clearok(curscr, TRUE);
clrtobot();
refresh();
clearok(curscr, FALSE);
#else
printf("\nerase is now a inoperative fuction\n");
#endif
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

display()
{ char value;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In display  P=%ld s=%ld t=%ld %c\n",p, s, t,
          st[s]);
fclose(fp);
#endif
value = st[s];
writex(value);
s -= 1;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}
```

```
acceptx()
{ integer addr; char temp;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In acceptx P=%ld s=%ld t=%ld \n",p, s, t);
fclose(fp);
#endif
addr = st[s--];
readx(&temp);
st[addr] = temp;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

/*******************************************************/
/* This procedure has been rewritten.                  */
/* Instead of working with a directly printer, the     */
/* character to be printed is sent to a file named     */
/* printerfile.                                        */
/*******************************************************/
print()
{ FILE *fp, *fopen();

#if TRACE >= 1
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In print P=%ld s=%ld t=%ld\n",p, s, t);
fclose(fp);
#endif

fp = fopen("printerfile", "a");
fprintf(fp,"%c",st[s]);
fclose(fp);


s--;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}
```

```
/************************************************************
Here is the new version of the procedure get.


proc k_read_page(title: overname; drive, pageno: int;
file_size: overposition; var block: overpage);
parameters at procedure entry:
st[s -  0] : block      address of block accepting page
st[s -  1] : file_size.words (int)
st[s -  2] : file_size.pages (int)
st[s -  3] : pageno     (int)
st[s -  4] : driveno    (int)
st[s - 16] : title      array [1:12] (char)
************************************************************/
get()
{ char fullname[94], fname[13], word0, word1;
FILE *fp, *fopen();
integer words, wordsread=0, addr;

#if TRACE >= 1
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In get P=%ld s=%ld t=%ld\n",p, s, t);
fclose(fp);
#endif
s -= 17;

  /* Get the complete path and name of the file */
getfilename(fname, s + 1);
sprintf(fullname, "%s/%s", &drive[st[s+13]][0], fname);

  /* Open and seek the correct page of the file */
fp = fopen(fullname, "r");
fseek(fp, (st[s + 14] - 1) * (PAGESIZE * 2), 0);

  /* Determine the number of words (two byte words) to read
*/
words = (st[s + 14] < st[s + 15]) ? PAGESIZE : st[s + 16];
            /*(pageno < # pages in file) ? 512 : # words on
last page*/

  /* addr is where the page is to be read to */
addr = st[s + 17];

  /* Read the correct number of words */
while (wordsread < words)
   { integer temp;
   fscanf(fp, "%c%c", &word0, &word1);
   temp = 0xff & ((int)(word0));
   temp = (temp << 8) | (0xff & ((int)(word1)));
   st[addr + wordsread++] = (temp & 0x8000) ? (temp |
0xffff0000) : temp;
   }
fclose(fp);

  /* Fill the rest the page with FILLERWORD.         */
  /* This was originally used for debugging purposes. */
while (wordsread < PAGESIZE)
   st[addr + wordsread++] = FILLERWORD;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

/************************************************************
Here is the new version of the procedure put.

proc k_write_page(title: overname; drive, pageno: int;
file_size: overposition; var block: overpage);
parameters at procedure entry:
st[s -  0] : block      address of block holding page
st[s -  1] : file_size.words (int)
st[s -  2] : file_size.pages (int)
st[s -  3] : pageno      (int)
st[s -  4] : driveno     (int)
st[s - 16] : title      array [1:12] (char)

************************************************************/
put()
{ char fullname[94], fname[13], word[2];
FILE *fp, *fopen();
integer words, wordsput = 0, addr;

#if TRACE >= 1
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In put P=%ld s=%ld t=%ld\n",p, s, t);
fclose(fp);
#endif
s -= 17;

  /* Get the complete path and name of the file */
getfilename(fname, s + 1);
sprintf(fullname, "%s/%s", &drive[st[s+13]][0], fname);

  /* Open and seek the correct page of the file */
```

```
fp = fopen(fullname, "r+");
fseek(fp, (st[s + 14] - 1) * (PAGESIZE * 2), 0);

  /* Determine the number of words (two byte words) to write
*/
words = (st[s + 14] < st[s + 15]) ? PAGESIZE : st[s + 16];
            /*(pageno < # pages in file) ? 512 : # words on
last page*/

  /* addr is where the page is to be written from */
addr = st[s + 17];

  /* Write the correct number of words */
while (wordsput < words)
   { word[1] = st[addr + wordsput];
   word[0] = st[addr + wordsput++] >> 8;
   fprintf(fp,"%c%c", word[0], word[1]);
   }
fclose(fp);
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

/************************************************************
This is a new procedure to be passed to the operating system
as a procedure parameter.  It is used to determine if a
directory input by the user is a valid directory.  The
parameter 'value' in the following sample call is returned
TRUE if the directory exists and the user has access to it,
and FALSE otherwise.

proc k_checkdrive(drivedir: overline; drive: int;
                  var value: bool);
parameters at procedure entry:
st[s -  0] : value     *  address of bool indicating if   *
                       *  drivedir is a valid directory   *
st[s -  1] : drive     *  (int)    drive number (1 or 2)  *
st[s - 81] : drivedir  *  array [1:81] (char)
************************************************************/
check_drive()
{ integer x=0, driveno;
char c, *fname, line[100];
FILE *fp, *fopen();

#if TRACE >= 1
fp = fopen("log","a");
```

```
if (printit)
   fprintf(fp, "In check_drive P=%ld s=%ld t=%ld\n",p, s,
          t);
fclose(fp);
#endif
s -= 82;
c = st[s + 1];
driveno = st[s + 81]; /* get the drive number */

  /* Copy the directory to be checked into a string */
while ((x <= 80) && (    ((c >= 'a') && (c <= 'z'))
   || ((c >= '-') && (c <= '9'))
   || ((c >= 'A') && (c <= 'Z'))
   ||   c == '/'  ||  c == '\\' || c == ':'))
   { drive[driveno][x++] = c;
   c = st[s + 1 + x];
   }
#ifdef UNIXSYSTEM
if (drive[driveno][x-1] != '/')
   drive[driveno][x++] = '/';
#endif
#ifdef MSDOS
if (drive[driveno][x-1] != '\\')
   drive[driveno][x++] = '\\';
#endif

drive[driveno][x] = '\0';

/* Get a temporary file and direct a directory listing    */
/* for the directory to be checked into the temporary file*/
gettempfile(&fname);
#ifdef MSDOS
sprintf(systemcall, "dir %s > %s", &drive[driveno][0],
fname);
#else
sprintf(systemcall, "ls -l %s > %s 2>&1",
&drive[driveno][0], fname);
#endif
#if TRACE < 0
printf("%s - Called from check_drive()\n",systemcall);
#endif
system(systemcall);

  /* Open the temporary file and read the first line */
fp = fopen(fname, "r");
#ifdef MSDOS
fscanf(fp, "%[^\n]s", line);
fscanf(fp, "%[^\n]s", line);
fscanf(fp, "%[^\n]s", line);
fscanf(fp, "%[^\n]s", line);
fscanf(fp, "%[^\n]s", line);
if (feof(fp))
   st[st[s + 82]] = FALSE;
else
   st[st[s + 82]] = TRUE;
#else
fscanf(fp, "%[^\n]s", line);

/* If the directory exists and the user has access, the  */
/* first line will have "total " for the first six
   characters */
if (strncmp(line, "total ", 6))
   st[st[s + 82]] = FALSE;
else
   st[st[s + 82]] = TRUE;
#endif
  /* close and remove the temporary file */
fclose(fp);
#ifdef MSDOS
sprintf(systemcall, "erase %s > NUL", fname);
#else
sprintf(systemcall, "rm %s", fname);
#endif
#if TRACE < 0
printf("%s - Called from check_drive()\n",systemcall);
#endif
system(systemcall);
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

/**********************************************************
 This is a new procedure to be passed to the operating
system as a procedure parameter.  It is used to copy files.


proc k_copy_file(file1, file2: overname; drive1,
                 drive2: int);

parameters at procedure entry:
st[s -  0] : drive2      (int)
st[s -  1] : drive1      (int)
st[s - 13] : file2       array [1:12] (char)
st[s - 25] : file1       array [1:12] (char)

**********************************************************/
```

```
copy_file()
{ char file1[13], file2[13];

#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In copy_file P=%ld s=%ld t=%ld\n",p, s, t);
fclose(fp);
#endif
s -= 26;

  /* if the first file is "*" then copy all */
  /* files from directory 1 to directory 2  */
if (st[s + 1] == '*')
#ifdef MSDOS
   sprintf(systemcall, "copy %s* %s > NUL", &drive[st[s +
25]][0],
   &drive[st[s + 26]][0]);
#else
sprintf(systemcall, "cp %s* %s", &drive[st[s + 25]][0],
&drive[st[s + 26]][0]);
#endif
else
   { getfilename(file1, s + 1);
   getfilename(file2, s + 13);
#ifdef MSDOS
   sprintf(systemcall, "copy %s%s %s%s > NUL", &drive[st[s +
25]][0], file1,
   &drive[st[s + 26]][0], file2);
#else
   sprintf(systemcall, "cp %s%s %s%s", &drive[st[s +
25]][0], file1,
   &drive[st[s + 26]][0], file2);
#endif
   }
#if TRACE < 0
printf("%s - Called from copy_file()\n",systemcall);
#endif
system(systemcall);
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

/*************************************************************
This is a new procedure to be passed to the operating
```

system as a procedure parameter. It is used to get the directory of a directory which is used as a virtual foppy disk.

```
proc k_get_diskcatalog(var catalog: overcatalog;
                            drive: int);
parameters at procedure entry:

st[s -  0] : drive      (int)
st[s -  1] : catalog    address of catalog variable

The catalog has the form:
size: int;
contents: table;
unused: filler;

table has the form:
array [1:47] (item)

item has the form:
title: name;
attr: attributes

attributes has the form:
address: int;
length: position;
protected: bool;

position has the form:
pages, words: int;
```

So from a base address of base, the overall structure on the stack looks like:

```
st[base +  0] : size     "number of item entries in catalog"

item1base = base + 1
st[item1base +  0] : title
st[item1base + 12] : attr.address
     "address is not needed without real disks to map"
st[item1base + 13] : attr.position.pages "# of pages"
st[item1base + 14] : attr.position.words
                                 "# words on last page"
st[item1base + 15] : attr.protected
item2base = item1base + 16
st[item2base +  0] : title
st[item2base + 12] : attr.address
         "address is not needed without real disks to map"
st[item2base + 13] : attr.position.pages "# of pages"
st[item2base + 14] : attr.position.words
                                 "# words on last page"
```

```
st[item2base + 15] : attr.protected
item47base = item46base + 16
**************************************************************/
#ifdef MSDOS                    /* MS-DOS version  of
get_diskcatalog */
get_diskcatalog()
{ long itemaddr, value, count=0;
FILE *fp, *fopen();
char *fname, line[100], *ptr, *login;
integer w;

#if TRACE >= 1
fp = fopen("log","a");
if (printit) fprintf(fp, "In get_diskcatalog P=%ld s=%ld
          t=%ld\n",p, s, t);
fclose(fp);
#endif
s -= 2;
itemaddr = st[s + 1] + 1;
gettempfile(&fname);
w = strlen(drive[st[s+2]][0]);
strncat(strcpy(systemcall, "dir "), &drive[st[s + 2]][0],w);
strcat(strcat(systemcall, " > "), fname);
#if TRACE < 0
printf("%s - Called from get_diskcatalog()\n",systemcall);
#endif
system(systemcall);
fp = fopen(fname, "r");
   for (w=1; w<5; w++) {      /*  Remove the first 4
                                  lines of the directory */
   fscanf(fp,"%[^\n]s",line);
   fgetc(fp);
   }
while (!(feof(fp)))
   { fscanf(fp,"%[^\n]s",line);
   fgetc(fp);
   if ((count < 47) &&  (strncmp(&line[10],"File(s)",7)) &&
      (strncmp(&line[13],"<DIR>",5)) &&
      (!(feof(fp))) && (strcmp(fname, &line[0])))
      {
      for (w=0; line[w]!='\0'; w++)
          /* Make all files lower case */
         line[w] = tolower(line[w]);
      st[itemaddr + 15] = FALSE;
      ptr = &line[13];
      value = 0;
      while ((*ptr < '0') || (*ptr > '9'))
         ptr++;
      while ((*ptr >= '0') && (*ptr <= '9'))
         { value = value * 10 + (*ptr - '0');
```

```
            ptr++;
            }
         value = value / 2;
         st[itemaddr + 13] = value / PAGESIZE + 1;
         if (!(st[itemaddr + 14] = value % PAGESIZE))
            { st[s + 13] -= 1;
            st[s + 14] = PAGESIZE;
            }
         value = 0;
         while ((value < 12) && (line[0 + value] != '\0'))
            { st[itemaddr + value] = line[0 + value];
            value++;
            }
         while (value < 12)
            st[itemaddr + value++] = ' ';
         count++;
         itemaddr += 16;
         }
      }
st[st[s + 1]] = count;
fclose(fp);
strcat(strcpy(systemcall, "erase "), fname);
#if TRACE < 0
printf("%s - Called from get_diskcatalog()\n",systemcall);
#endif
system(systemcall);
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}
#endif

#ifdef UNIXSYSTEM

/* UNIXSYSTEM Version of get_diskcatalog() */

get_diskcatalog()
{ integer itemaddr, value, count=0;
FILE *fp, *fopen();
char *fname, line[100], *ptr, *login;

#if TRACE >= 1
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In get_diskcatalog P=%ld s=%ld t=%ld\n",p,
         s, t);
fclose(fp);
```

```
#endif
s -= 2;

  /* find the name of the user */
login = getpwuid(getuid())->pw_name;

  /* set itemaddr to point to the first item in catalog */
itemaddr = st[s + 1] + 1;

/* Get a temporary file and direct a directory listing */
/* for the directory to be checked into the temporary file*/
gettempfile(&fname);
sprintf(systemcall, "ls -l %s > %s", &drive[st[s + 2]][0],
fname);
#if TRACE < 0
printf("%s - Called from get_diskcatalog()\n",systemcall);
#endif
system(systemcall);

  /* Open the temporary file and skip over the first line */
fp = fopen(fname, "r");
fscanf(fp,"%[^\n]s",line);
fgetc(fp);

  /* read in the first line of the directory listing */
fscanf(fp,"%[^\n]s",line);
fgetc(fp);

/*************************************************************
for each line in the directory listing put the
file in the catalog if it meets certain criteria
criteria :
catalog only holds 47 files. Each line must represent a
file.   "line[0] == '-' represents a file. The filename must
be less or equal to 12 characters in length.  Don't include
the kernel's temporary file.  Files are ignored if the user
does not have read access.
*************************************************************/
while (!(feof(fp)))
   { if ((count < 47)             &&
      (line[0] == '-')            &&
      (strlen(&line[45]) <= 12)   &&
      (strcmp(fname, &line[45])) &&
      ((strncmp(&line[14], login, strlen(login)) ?
      (line[7] == 'r') : (line[1] == 'r')))
      {
      if (strncmp(&line[14], login, strlen(login)) == 0)
         st[itemaddr + 15] = (line[2] == '-');
      else
         st[itemaddr + 15] = (line[8] == '-');
         /* read the size of the file in bytes */
         ptr = &line[20];
         value = 0;
         while ((*ptr < '0') || (*ptr > '9'))
            ptr++;
         while ((*ptr >= '0') && (*ptr <= '9'))
            { value = value * 10 + (*ptr - '0');
            ptr++;
            }

         /* convert from bytes to words (two byte words) */
         value = value / 2;

         /* determine number of pages and number of     */
         /* words on last page from the number of words */
         st[itemaddr + 13] = value / PAGESIZE + 1;
         if (!(st[itemaddr + 14] = value % PAGESIZE))
            { st[s + 13] -= 1;
            st[s + 14] = PAGESIZE;
            }

         /* copy the filename */
         value = 0;
         while ((value < 12) && (line[45 + value] != '\0'))
            { st[itemaddr + value] = line[45 + value];
            value++;
            }

         /* fill the unused part of the name with blanks */
         while (value < 12)
            st[itemaddr + value++] = ' ';

         /* increment the count and point to the next item */
         count++;
         itemaddr += 16;
         }

       /* read the next line */
      fscanf(fp,"%[^\n]s",line);
      fgetc(fp);
      }

    /* put the number of items in the catalog */
   st[st[s + 1]] = count;

    /* close and remove the temporary file */
   fclose(fp);
   sprintf(systemcall, "rm %s", fname);
   #if TRACE < 0
```

```
printf("%s - Called from get_diskcatalog()\n",systemcall);
#endif
system(systemcall);
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}
#endif       /* UNIXSYSTEM Version */

/************************************************************
 This is a new procedure to be passed to the operating
system as a procedure parameter.  It is used to delete
files.  If the file to be removed is 'halt.command' and the
directory  number is number 5 (an invalid drive), then the
kernel  exits gracefully (ie the user has typed in the
command 'exit').


proc k_rm_file(file1: overname; drive1: int);
parameters at procedure entry:
   st[s -  0] : drive1     (int)
st[s - 12] : file1      array [1:12] (char)
*************************************************************/
rm_file()
{ char file[13];

#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In rm_file P=%ld s=%ld t=%ld \n", p, s, t);
fclose(fp);
#endif
s -= 13;
getfilename(file, s + 1);
if ((strcmp(file, "halt.command") == 0) && (st[s+13] == 5))
   quit(0);
#ifdef MSDOS
sprintf(systemcall, "erase %s%s > NUL", &drive[st[s+13]][0],
file);
#endif
#ifdef UNIXSYSTEM
sprintf(systemcall, "rm %s%s", &drive[st[s+13]][0], file);
#endif
#if TRACE < 0
printf("%s - Called from rm_file()\n",systemcall);
#endif
```

```
system(systemcall);
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

/**********************************************************
This is a new procedure to be passed to the operating system
as a procedure parameter. It is used to protect and unpro-
tect files.

proc k_protect_file(title: overname; drive: int;
                   value: bool))
parameters at procedure entry:
st[s -  0] : value      (bool)
TRUE:protect  FALSE:unprotect
st[s -  1] : drive      (int)
st[s - 13] : title      array [1:12] (char)
**********************************************************/
protect_file()
{ char file[13];

#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In protect_file P=%ld s=%ld t=%ld\n",p, s,
          t);
fclose(fp);
#endif
s -= 14;
getfilename(file, s + 1);
#ifdef MSDOS   /* Requires MS-DOS version 3.x  ATTRIB.EXE */
sprintf(systemcall, "attrib %s %s%s", (st[s+ 14] ? "+R" :
"-R"),
&drive[st[s+13]][0], file);
#endif
#ifdef UNIXSYSTEM
sprintf(systemcall, "chmod %s %s%s", (st[s + 14] ? "ugo-w" :
"u+w"),
&drive[st[s+13]][0], file);
#endif
#if TRACE < 0
printf("%s - Called from protect_file()\n",systemcall);
#endif
system(systemcall);
#if TRACE >= 2
```

```
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

kernelcall(procno)
integer procno;
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In kernelcall  P=%ld s=%ld t=%ld %ld\n",p,
          s, t, procno);
fclose(fp);
#endif
#if TRACE >= 3
stack_dump(85);
#endif
switch(procno)
   {
   case CURSORNO:
      cursor();
      break;
   case ERASENO:
      kerase();
      break;
   case DISPLAYNO:
      display();
      break;
   case ACCEPTNO:
      acceptx();
      break;
   case PRINTNO:
      print();
      break;
   case GETNO:
      get();
      break;
   case PUTNO:
      put();
      break;
   case CHECKDRIVENO:
      check_drive();
      break;
   case COPYNO:
      copy_file();
      break;
   case GETCATALOGNO:
      get_diskcatalog();
      break;
   case RMNO:
      rm_file();
      break;
   case PROTECTNO:
      protect_file();
      break;
   }
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

/* standard instructions */

newline(lineno)
{ p += 2;
}

gotox(displ)
integer displ;
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In gotox  P=%ld s=%ld t=%ld %ld   destina-
         tion=%ld\n",
   p, s, t, displ, p+(displ/2));
fclose(fp);
#endif
p = p + (displ / 2);
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}

libproc(paramlength,
templength,lineno)
integer paramlength,templength,lineno;
{
#if TRACE >= 1
```

```
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In libproc  P=%ld s=%ld t=%ld %ld %ld
          %ld\n",
   p, s, t, paramlength, templength, lineno);
fclose(fp);
#endif
if (tasks>1)
   callerror(lineno);

st[b + 2] =
b - (paramlength / 2) - 1;
if (s + (templength / 2) > t)
   variablelimit(lineno);

p = p + 4;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

endlib(lineno)
integer lineno;
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In endlib  P=%ld s=%ld t=%ld %ld\n",p, s, t,
          lineno);
fclose(fp);
#endif
moveprogram();
p = t + NAMELENGTH + 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}


procedure(paramlength,varlength,
templength, lineno)
integer paramlength,varlength;
integer templength, lineno;
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In procedure  P=%ld s=%ld t=%ld b=%ld %ld
          %ld %ld %ld\n",
   p, s, t, b, paramlength, varlength, templength, lineno);
fclose(fp);
#endif
st[b + 2] =
b - (paramlength / 2) - 1;
s = s + (varlength / 2);
if (s + (templength / 2) > t)
   variablelimit(lineno);

p = p + 5;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(3);
#endif
}

endproc()
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In endproc P=%ld s=%ld t=%ld b=%ld\n",p, s,
          t, b);
fclose(fp);
#endif
   if (st[b + 4] != NONE) {
   p = st[b + 4];
   t = st[b + 3];
   s = st[b + 2];
   b = st[b + 1]; }
else
   p = p + 1;

#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
```

```
}

field(displ)
integer displ;
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In field  P=%ld s=%ld t=%ld %ld\n",
   p, s, t, displ);
fclose(fp);
#endif
st[s] = st[s] + (displ / 2);
p = p + 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(5);
#endif
}

indexx(lower,upper,length,lineno)
integer lower,upper,length,lineno;
{
integer i;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In indexx  P=%ld s=%ld t=%ld %ld %ld %ld
         %ld\n",
   p, s, t,lower, upper, length, lineno);
fclose(fp);
#endif
i = st[s];
s = s - 1;
if (i<lower || i>upper)

   rangeerror(lineno);

st[s] = st[s] +
(i - lower) * (length / 2);
p = p + 5;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(5);
```

```
#endif
}

instance(steps)
integer steps;
{
integer link,m;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In instance  P=%ld s=%ld t=%ld %ld\n",p, s,
          t, steps);
fclose(fp);
#endif
link = b;
m = steps;
   while (m > 0) {
   link = st[link];
   m = m - 1;
   }
s = s + 1;
st[s] = link;
p = p + 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(5);
#endif
}

variable(displ)
integer displ;
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In variable  P=%ld s=%ld t=%ld %ld\n",p, s,
          t, displ);
fclose(fp);
#endif
st[s] = st[s] + (displ / 2);
p = p + 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(5);
```

```
#endif
}


blank(number)
integer number;
{
integer i;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In blank  P=%ld s=%ld t=%ld %ld\n",p, s, t,
          number);
fclose(fp);
#endif
i = 0;
   while (i < number) {
   i = i + 1;
   st[s + i] = SP;
   }
s = s + number;
p = p + 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}

construct(number,lineno)
integer number,lineno;
{ integer member, i;
settype1 new;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In construct  P=%ld s=%ld t=%ld %ld
          %ld\n",p, s, t,number, lineno);
fclose(fp);
#endif
for (i=0; i<SETLENGTH; i++)
   new.settype2[i] = 0;
i = 0;
while (i < number)
   { member = st[s--];
   if ((member < 0) ||
      (member > SETLIMIT))
      rangeerror(lineno);



   insert_member(new.settype2,
   member);
   i++;
   }
storeset(s + 1, &new);
s += SETLENGTH;
p += 3;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(15);
#endif
}


constant(value)
integer value;
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In constant  P=%ld s=%ld t=%ld %ld\n",p, s,
          t, value);
fclose(fp);
#endif
s = s + 1;
st[s] = value;
p = p + 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}

value(length)
integer length;
{
integer y,i;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
```

```
   fprintf(fp, "In value  P=%ld s=%ld t=%ld %ld\n",p, s, t,
           length);
fclose(fp);
#endif
length = length / 2;
y = st[s];  i = 0;
   while (i < length) {
   st[s + i] = st[y + i];
   i = i + 1;
   }
s = s + length - 1;
p = p + 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(length+3);
#endif
}

valspace(length)
integer length;
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In valspace  P=%ld s=%ld t=%ld %ld\n",p, s,
           t, length);
fclose(fp);
#endif
s = s + (length / 2);
p = p + 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(1);
#endif
}

notx()
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In notx  P=%ld s=%ld t=%ld \n",p, s, t);
fclose(fp);
#endif
```

```
st[s] = (! (st[s]));
p = p + 1;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}


multiply(lineno)
integer lineno;
{ long sts, sts_1;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In multiply  P=%ld s=%ld t=%ld %ld\n",p, s,
          t, lineno);
fclose(fp);
#endif
s--;
sts = st[s]; sts_1 = st[s+1];
st[s] = st[s] * st[s + 1];

if (sts * sts_1 != st[s])
   rangeerror(lineno);

p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}

divide(lineno)
integer lineno;
{ long sts, sts_1;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In divide  P=%ld s=%ld t=%ld %ld\n",p, s, t,
          lineno);
fclose(fp);
#endif
s--;
```

```
sts = st[s]; sts_1 = st[s+1];
st[s] = st[s] / st[s + 1];

if (sts / sts_1 != st[s])
   rangeerror(lineno);

p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}

modulo(lineno)
integer lineno;
{ long sts, sts_1;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In modulo  P=%ld s=%ld t=%ld %ld\n",p, s, t,
          lineno);
fclose(fp);
#endif
s--;
sts = st[s]; sts_1 = st[s+1];
st[s] = st[s] % st[s + 1];

if (sts % sts_1 != st[s])
   rangeerror(lineno);

p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}

andx(){
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In andx  P=%ld s=%ld t=%ld \n",p, s, t);
fclose(fp);
#endif
```

```
s--;
st[s] = st[s] && st[s + 1];
p++;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}

intersection()
{ settype1 x, y;
integer i;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In intersection P=%ld s=%ld t=%ld \n",p, s,
          t);
fclose(fp);
#endif

s -= SETLENGTH;
loadset(s + 1, &y);
loadset(s - SETLENGTH + 1, &x);

for (i=0; i<SETLENGTH; i++)
   x.settype2[i] = x.settype2[i] & y.settype2[i];
storeset(s - SETLENGTH + 1, &x);

p += 1;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


minus(lineno)
integer lineno;
{ long sts;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In minus  P=%ld s=%ld t=%ld %ld\n",p, s, t,
          lineno);
```

```
fclose(fp);
#endif
sts = st[s];
st[s] = - st[s];
if (-sts != st[s])
   rangeerror(lineno);

p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}


add(lineno)
integer lineno;
{ long sts, sts_1;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In add   P=%ld s=%ld t=%ld %ld\n",p, s, t,
           lineno);
fclose(fp);
#endif
s--;
sts = st[s]; sts_1 = st[s+1];
st[s] = st[s] + st[s + 1];

if (sts + sts_1 != st[s])
   rangeerror(lineno);

p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}

subtract(lineno)
integer lineno;
{ long sts, sts_1;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
```

```
if (printit)
   fprintf(fp, "In subtract  P=%ld s=%ld t=%ld %ld\n",p, s,
          t, lineno);
fclose(fp);
#endif
s--;
sts = st[s]; sts_1 = st[s+1];
st[s] = st[s] - st[s + 1];

if (sts - sts_1 != st[s])
   rangeerror(lineno);

p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}

orx(){
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In orx   P=%ld s=%ld t=%ld \n",p, s, t);
fclose(fp);
#endif
s--;
st[s] = st[s] || st[s + 1];

p++;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}

unionx()
{ settype1 x, y;
integer i;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In union P=%ld s=%ld t=%ld \n",p, s, t);
fclose(fp);
```

```
#endif

s -= SETLENGTH;
loadset(s + 1, &y);
loadset(s - SETLENGTH + 1, &x);

for (i=0; i<SETLENGTH; i++)
   x.settype2[i] = x.settype2[i] | y.settype2[i];
storeset(s - SETLENGTH + 1, &x);

p += 1;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}

difference()
{ settype1 x, y;
integer i;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In difference P=%ld s=%ld t=%ld \n",p, s,
          t);
fclose(fp);
#endif

s -= SETLENGTH;
loadset(s + 1, &y);
loadset(s - SETLENGTH + 1, &x);

for (i=0; i<SETLENGTH; i++)
   x.settype2[i] = x.settype2[i] & ~ y.settype2[i];
storeset(s - SETLENGTH + 1, &x);

p += 1;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


equal(length)
```

```
integer length;
{ integer y, i = 0;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In equal  P=%ld s=%ld t=%ld %ld\n",p, s, t,
          length);
fclose(fp);
#endif
length = length / 2;
y = s - length + 1;
s = y - length;

while ((i < length - 1) &&
   (st[s + i] == st[y + i]))

   i++;

st[s] = st[s + i] == st[y + i];

p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}

notequal(length)
integer length;
{ integer y, i = 0;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In notequal  P=%ld s=%ld t=%ld %ld\n",p, s,
          t, length);
fclose(fp);
#endif

length = length / 2;
y = s - length + 1;
s = y - length;

while ((i < length - 1) &&
   (st[s + i] == st[y + i]))

   i++;
```

```
st[s] = st[s + i] != st[y + i];


p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}

less()
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In less P=%ld s=%ld t=%ld \n",p, s, t);
fclose(fp);
#endif
s--;
st[s] = st[s] < st[s + 1];

p++;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}

notless()
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In notless P=%ld s=%ld t=%ld \n",p, s, t);
fclose(fp);
#endif
s--;
st[s] = st[s] >= st[s + 1];

p++;
#if TRACE >= 2
register_dump();
#endif
```

```
#if TRACE >= 3
stack_dump(2);
#endif
}

greater()
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In greater P=%ld s=%ld t=%ld \n",p, s, t);
fclose(fp);
#endif
s--;
st[s] = st[s] > st[s + 1];

p++;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}

notgreater()
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In notgreater P=%ld s=%ld t=%ld \n",p, s,
          t);
fclose(fp);
#endif
s--;
st[s] = st[s] <= st[s + 1];

p++;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}

inx(lineno)
integer lineno;
```

```
{ integer x, displ, x_map;
settype1 y;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In inx   P=%ld s=%ld t=%ld %ld\n",p, s, t,
          lineno);
fclose(fp);
#endif

s -= SETLENGTH;
loadset(s + 1, &y);
x = st[s];
if ((x < 0) || (x > SETLIMIT))
   rangeerror(lineno);

displ = (x & 0x70) >> 4;
x_map = 0x8000 >> (x & 0xf);
st[s] = x_map & y.settype2[displ];
p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}


assign(length)
integer length;
{ integer x, y, i = 0;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In assign P=%ld s=%ld t=%ld %ld\n",p, s,
          t,length);
fclose(fp);
#endif
length = length / 2;
s = s - length - 1;
x = st[s + 1];
y = s + 2;

while (i < length)
   { st[x + i] = st[y + i];
   i++;
   }
```

```
p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}


addrx(){
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In addrx P=%ld s=%ld t=%ld \n",p, s, t);
fclose(fp);
#endif
s--;
st[s] = st[s + 1];
p++;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(2);
#endif
}

haltx(lineno)
integer lineno;{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In haltx  P=%ld s=%ld t=%ld %ld\n",p, s, t,
          lineno);
fclose(fp);
#endif
stop(lineno, "halt#");
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}

obtainx(){
#if TRACE >= 1
```

```
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In obtainx P=%ld s=%ld t=%ld \n",p, s, t);
fclose(fp);
#endif
s -= 2;
if ((st[s + 2] < 0) ||
   (st[s + 2] > (MAXADDR + NUMCOMMCHANNELS + 1)))
   stop(linenumber, "obtain address out of range#");
else
   st[st[s + 2]] =
   st[s + 1];
p++;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}

placex(){
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In placex P=%ld s=%ld t=%ld \n",p, s, t);
fclose(fp);
#endif
s -= 2;
if ((st[s + 1] < 0) ||
   (st[s + 1] > (MAXADDR + NUMCOMMCHANNELS + 1)))
   stop(linenumber, "place address out of range#");
else
   st[st[s + 1]] = st[s + 2];

p++;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}

sensex(){
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
```

```
if (printit)
   fprintf(fp, "In sensex P=%ld s=%ld t=%ld \n",p, s, t);
fclose(fp);
#endif

s -= 2;
if ((st[s + 1] < 0) ||
   (st[s + 1] > (MAXADDR + NUMCOMMCHANNELS + 1)))
   stop(linenumber, "place address out of range#");
else
   st[s] =
   st[s + 1] &
   st[s + 2];
p++;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


procarg(displ)
integer displ;
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In procarg  P=%ld s=%ld t=%ld %ld\n",p, s,
          t, displ);
fclose(fp);
#endif
st[++s] = p + (displ / 2);
p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}

paramarg(displ)
integer displ;
{ integer addr;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
```

```
if (printit)
   fprintf(fp, "In paramarg  P=%ld s=%ld t=%ld %ld\n",p, s,
          t, displ);
fclose(fp);
#endif
addr = st[s] + (displ / 2);
st[s] = st[addr];
st[s + 1] = st[addr + 1];
s++;
p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


proccall(displ)
integer displ;{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In proccall  P=%ld s=%ld t=%ld %ld\n",p, s,
          t,displ);
fclose(fp);
#endif
st[s + 1] = b;
st[s + 3] = t;
st[s + 4] = p + 2;
b = s;
s += 4;
p += (displ / 2);
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}

paramcall(displ)
integer displ;
{ integer addr, dest;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
```

```
   fprintf(fp, "In paramcall  P=%ld s=%ld t=%ld %ld\n",p, s,
          t, displ);
fclose(fp);
#endif
addr = st[s] + (displ / 2);
dest = st[addr + 1];
if (dest <= MAXPARAM)
   { s--;
   kernelcall(dest);
   p += 2;
   }
else
   { st[s] = st[addr];
   st[s + 1] = b;
   st[s + 3] = t;
   st[s + 4] = p + 2;
   b = s;
   s += 4;
   p = dest;
   }
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


dox(displ)
integer displ;{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In dox   P=%ld s=%ld t=%ld %ld\n",p, s, t,
          displ);
fclose(fp);
#endif
if (st[s])
   p += 2;
else
   p += (displ / 2);

s--;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
```

```
#endif
}

elsex(displ)
integer displ;
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In elsex  P=%ld s=%ld t=%ld %ld\n",p, s, t,
          displ);
fclose(fp);
#endif
p += (displ / 2);
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


whenx()
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In whenx  P=%ld s=%ld t=%ld\n",p, s, t);
fclose(fp);
#endif
p++;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}

waitx(displ)
integer displ;{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In waitx  P=%ld s=%ld t=%ld %ld\n",p, s, t,
          displ);
```

```
fclose(fp);
#endif
p += (displ / 2);
preempt();
this = this % tasks + 1;
resume();
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}

endwhen()
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In endwhen  P=%ld s=%ld t=%ld\n", p, s, t);
fclose(fp);
#endif
p++;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


process(templength, lineno)
integer templength, lineno;
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In process  P=%ld s=%ld t=%ld %ld %ld\n",
   p, s, t, templength, lineno);
fclose(fp);
#endif
if (s + (templength / 2) > t)
   variablelimit(lineno);

p += 3;
#if TRACE >= 2
register_dump();
```

```
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}

alsox(displ)
integer displ;{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In alsox  P=%ld s=%ld t=%ld %ld\n",p, s, t,
          displ);
fclose(fp);
#endif
if (tasks > 1)
   { while (this < tasks)
      { q[this] = q[this + 1];

      this++;
      }
   tasks -= 1;
   this = 1;
   resume();
   } else if (tasks == 1)
   { s = stacktop;
   t = progtop;
   p += (displ / 2);
   }
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


cobeginx(number, lineno)
integer number, lineno;
{ integer length, i;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In cobeginx  P=%ld s=%ld t=%ld %ld %ld\n",p,
          s, t, number, lineno);
fclose(fp);
#endif
```

```
tasks = number;
if (tasks > MAXPROC)
   processlimit(lineno);

stacktop = s;
progtop = t;
length = (t - s) / tasks;

i = 0;
while (i < tasks)
   { i++;
   t = s + length;
   q[i].bx = b;
   q[i].sx = s;
   q[i].tx = t;
   q[i].px = p + (st[p + 2 * i + 2] / 2);

   s = t;
   }
this = 1;
resume();
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


endcode(lineno)
integer lineno;{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In endcode  P=%ld s=%ld t=%ld %ld\n",p, s,
          t, lineno);
fclose(fp);
#endif
stop(lineno, "terminated#");
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
```

```
}



localvar(displ)
integer displ;{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In localvar  P=%ld s=%ld t=%ld b=%ld
          %ld\n",p, s, t, b, displ);
fclose(fp);
#endif
st[++s] = b + (displ / 2);
p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(5);
#endif
}

outervar(displ)
integer displ;{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In outervar  P=%ld s=%ld t=%ld %ld\n",p, s,
          t, displ);
fclose(fp);
#endif
st[++s] = st[b] + (displ / 2);
p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}

localvalue(displ)
integer displ;{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
```

```
   fprintf(fp, "In localvalue  P=%ld s=%ld t=%ld b=%ld
          %ld\n",p, s, t, b, displ);
fclose(fp);
#endif
st[++s] = st[b + (displ / 2)];
p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(5);
#endif
}


outervalue(displ)
integer displ;{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In outervalue  P=%ld s=%ld t=%ld %ld\n",p,
          s, t, displ);
fclose(fp);
#endif
st[++s] = st[st[b] + (displ / 2)];

p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


localset(displ)
integer displ;
{ integer addr, i = 0;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In localset  P=%ld s=%ld t=%ld %ld\n",p, s,
          t,displ);
fclose(fp);
#endif
addr = b + (displ / 2);
while (i < SETLENGTH)
```

```
    { st[s + i + 1] = st[addr + i];
    i++;
    }

s += SETLENGTH;
p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


outerset(displ)
integer displ;
{ integer addr, i = 0;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In outerset  P=%ld s=%ld t=%ld %ld\n",p, s,
          t,displ);
fclose(fp);
#endif

addr = st[b] + (displ / 2);
while (i < SETLENGTH)
   { st[s + i + 1] = st[addr + i];
   i++;
   }

s += SETLENGTH;
p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


localcase(vardispl,value,progdispl)
integer vardispl, value, progdispl;
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
```

```
if (printit)
   fprintf(fp, "In localcase  P=%ld s=%ld t=%ld %ld %ld
          %ld\n",p
   , s, t, vardispl, value, progdispl);
fclose(fp);
#endif
if (st[b + (vardispl / 2)] == value)
   p += 4;
else
   p += (progdispl / 2);


#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


outercase(vardispl,value,progdispl)
integer vardispl, value, progdispl;
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In outercase  P=%ld s=%ld t=%ld %ld %ld
          %ld\n",
   p, s, t, vardispl, value, progdispl);
fclose(fp);
#endif
if (st[st[b]+ (vardispl / 2)] == value)
   p += 4;
else
   p += (progdispl / 2);


#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


stringconst(number)
integer number;
```

```
{ integer i = 0;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In stringconst  P=%ld s=%ld t=%ld %ld\n",p,
          s, t, number);
fclose(fp);
#endif

while (i < number)
   { i++;
   st[s + i] = st[p + i + 1];

   }
s += number;
p += number + 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10+number);
#endif
}


setconst(number)
integer number;
{ integer i;
settype1 new;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In setconst  P=%ld s=%ld t=%ld %ld\n",p, s,
          t, number);
fclose(fp);
#endif
for (i=0; i<SETLENGTH; i++)
   new.settype2[i] = 0;
i = 0;
while (i++<number)
   { insert_member(new.settype2,
   st[p + i + 1]);

   }
storeset(s + 1, &new);
s += SETLENGTH;
p += number + 2;
#if TRACE >= 2
```

```
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


singleton(value)
integer value;
{ settype1 new;
integer i;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In singleton  P=%ld s=%ld t=%ld %ld\n",p, s,
          t, value);
fclose(fp);
#endif
for (i=0;i<SETLENGTH;i++)
   new.settype2[i] = 0;
insert_member(new.settype2, value);
storeset(s + 1, &new);
s += SETLENGTH;
p += 2;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


elemvalue(){
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In elemvalue P=%ld s=%ld t=%ld \n",p, s, t);
fclose(fp);
#endif
st[s] = st[st[s]];
p += 1;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
```

```
#endif
}


elemassign(){
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In elemassign P=%ld s=%ld t=%ld \n",p, s,
          t);
fclose(fp);
#endif
st[st[s - 1]] = st[s];
s -= 2; p += 1;
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}


outercall(displ)
integer displ;
{
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In outercall  P=%ld s=%ld t=%ld %ld\n",p, s,
          t, displ);
fclose(fp);
#endif
st[++s] = st[b];
st[s + 1] = b;
st[s + 3] = t;
st[s + 4] = p + 2;
b = s;
s += 4;
p += (displ / 2);
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}
```

```
outerparam(displ)
integer displ;
{ integer addr, dest;
#if TRACE >= 1
FILE *fp, *fopen();
fp = fopen("log","a");
if (printit)
   fprintf(fp, "In outerparam  P=%ld s=%ld t=%ld %ld\n",p,
           s, t, displ);
fclose(fp);
#endif
addr = st[b] + (displ / 2);
dest = st[addr + 1];
if (dest <= MAXPARAM)
   { kernelcall(dest);
   p += 2;
   }
else
   { st[++s] = st[addr];
   st[s + 1] = b;
   st[s + 3] = t;
   st[s + 4] = p + 2;
   b = s;
   s += 4;
   p = dest;
   }
#if TRACE >= 2
register_dump();
#endif
#if TRACE >= 3
stack_dump(10);
#endif
}

execute_instruction()

{ integer op;

op = st[p];
switch (op)
   { case ADD4: add(st[p + 1]); break;

   case ALSO4: alsox(st[p + 1]); break;

   case AND4: andx(); break;

   case ASSIGN4: assign(st[p + 1]); break;

   case BLANK4: blank(st[p + 1]); break;
```

```
    case COBEGIN4: cobeginx(st[p + 1], st[p + 2]);
       break;
    case CONSTANT4: constant(st[p + 1]);
       break;
    case CONSTRUCT4: construct(st[p + 1], st[p + 2]);
       break;

    case DIFFERENCE4: difference(); break;

    case DIVIDE4: divide(st[p + 1]); break;

    case DO4: dox(st[p + 1]); break;

    case ELSE4: elsex(st[p + 1]); break;

    case ENDCODE4: endcode(st[p + 1]); break;

    case ENDLIB4: endlib(st[p + 1]); break;

    case ENDPROC4: endproc(); break;

    case ENDWHEN4: endwhen(); break;

    case EQUAL4: equal(st[p + 1]); break;

    case FIELD4: field(st[p + 1]); break;

    case GOTO4: gotox(st[p + 1]); break;

    case GREATER4: greater(); break;

    case IN4: inx(st[p + 1]); break;

    case INDEX4: indexx(st[p + 1], st[p + 2], st[p + 3], st[p
+ 4]); break;


    case INSTANCE4: instance(st[p + 1]); break;

    case INTERSECTION4: intersection(); break;

    case LESS4: less(); break;
    case LIBPROC4: libproc(st[p + 1], st[p + 2], st[p + 3]);
break;


    case MINUS4: minus(st[p + 1]); break;

    case MODULO4: modulo(st[p + 1]); break;
    case MULTIPLY4: multiply(st[p + 1]); break;

    case NEWLINE4: newline(st[p + 1]); break;

    case NOT4: notx(); break;
    case NOTEQUAL4: notequal(st[p + 1]); break;

    case NOTGREATER4: notgreater(); break;

    case NOTLESS4: notless(); break;

    case OR4: orx(); break;
    case PARAMARG4: paramarg(st[p + 1]); break;

    case PARAMCALL4: paramcall(st[p + 1]); break;

    case PROCARG4: procarg(st[p + 1]); break;

    case PROCCALL4: proccall(st[p + 1]); break;

    case PROCEDURE4: procedure(st[p + 1], st[p + 2], st[p +
3], st[p + 4]);
       break;

    case PROCESS4: process(st[p + 1], st[p + 2]); break;

    case SUBTRACT4: subtract(st[p + 1]); break;

    case UNION4: unionx(); break;
    case VALSPACE4: valspace(st[p + 1]); break;

    case VALUE4: value(st[p + 1]); break;

    case VARIABLE4: variable(st[p + 1]); break;

    case WAIT4: waitx(st[p + 1]); break;

    case WHEN4: whenx(); break;
    case ADDR4: addrx(); break;
    case HALT4: haltx(st[p + 1]); break;

    case OBTAIN4: obtainx(); break;

    case PLACE4: placex(); break;
    case SENSE4: sensex(); break;
    case ELEMASSIGN4: elemassign(); break;

    case ELEMVALUE4: elemvalue(); break;
```

```
   case LOCALCASE4: localcase(st[p + 1], st[p + 2], st[p +
3]); break;


   case LOCALSET4: localset(st[p + 1]); break;

   case LOCALVALUE4: localvalue(st[p + 1]); break;

   case LOCALVAR4: localvar(st[p + 1]); break;

   case OUTERCALL4: outercall(st[p + 1]); break;

   case OUTERCASE4: outercase(st[p + 1], st[p + 2], st[p +
3]); break;


   case OUTERPARAM4: outerparam(st[p + 1]); break;

   case OUTERSET4: outerset(st[p + 1]); break;

   case OUTERVALUE4: outervalue(st[p + 1]); break;

   case OUTERVAR4: outervar(st[p + 1]); break;

   case SETCONST4: setconst(st[p + 1]); break;

   case SINGLETON4: singleton(st[p + 1]); break;

   case STRINGCONST4: stringconst(st[p + 1]); break;

   default:          printf("Invalid OP.  P=%ld s=%ld t=%ld
Value = %ld\n",
      p, s, t, op);
      register_dump();
      stack_dump(10);
      prog_stack_dump(9);
      reboot();
      break;
   }
}


/*******************************************************/
/* Here is the main procedure for the Edison kernel.   */
/*******************************************************/
main()
{
printit = TRUE;
initialize();
```

```
while (TRUE)
   { execute_instruction();
   }
}


/*******************************************************/
/* This procedure will dump a section of the stack.    */
/* The procedure receives one parameter which is the   */
/* number of stack locations below the location pointed*/
/* at by the p 'register' to be dumped.  (600 max)     */
/* This procedure was added for debugging purposes.    */
/*******************************************************/
stack_dump(depth)
integer depth;
{ integer limit = -1, temp, max_depth = 600;
FILE *fp, *fopen();
unsigned d;
fp = fopen("log", "a");

  /* verify the depth */
if (depth <= 0)
   depth = 1;
if ((depth > max_depth) && (s > max_depth))
   limit = s - max_depth;
if ((depth < max_depth) && (depth < s))
   limit = s - depth;

  /* print a header */
fprintf(fp, "\n");
fprintf(fp, "\n                       hex          integer
binary");

  /* dump the stack */
for (depth = s; depth > limit; depth--)
   { fprintf(fp, "\n    st[%5d] ==     %08x%12d     ",
   depth, st[depth], st[depth]);
   temp = st[depth];
   d = 0x80000000;
   while (d != 0)
      { if (temp & d)
         fprintf(fp, "1");
      else
         fprintf(fp, "0");
      d >>= 1;
      }
   }
fprintf(fp, "\n");
fclose(fp);
}
```

```
/*********************************************************/
/* This procedure will dump a the values of the          */
/* registers for the user to see in both decimal and hex.*/
/* This procedure was added for debugging purposes.      */
/*********************************************************/
register_dump()
{
printf("\n THIS = %5X   TASKS = %5X      STACKTOP = %5X
PROGTOP = %5X\n"
,this,tasks,stacktop,progtop);
printf(" B = %5X    S = %5X    T = %5X    P =
%5X\n",b,s,t,p);
printf("\n");
printf(" THIS = %5d   TASKS = %5d      STACKTOP = %5d
PROGTOP = %5d\n"
,this,tasks,stacktop,progtop);
printf(" B = %5d    S = %5d    T = %5d    P =
%5d\n",b,s,t,p);
}

/*********************************************************/
/* This procedure will insert a new set member into a    */
/* set.  This is done by bitwise ORing the new member    */
/* into the set.                                         */
/* This procedure was added for set manipulation.        */
/*********************************************************/
insert_member(set, member)
integer set[SETLENGTH], member;
{ integer displ;
unsigned member_map;

displ = (member & 0x70) >> 4;
member_map = 0x00008000 >> (member & 0xf);
set[displ] = set[displ] | member_map;
}


/*********************************************************/
/* This procedure receives a character string which is   */
/* returned with the name of a file name not in use to   */
/* be used as a temporary file.                          */
/* This procedure was added.                             */
/*********************************************************/
gettempfile(fname)
char **fname;
{ FILE *fopen(), *fp;
*fname = "tempa";
```

```
while (fp = fopen(*fname,"r"))
   { char c;
   fclose(fp);
   c = *(*fname + 4) + 1;
   if (c == 91)
      c += 2;
   if (c == 96)
      c++;
   *(*fname + 4) = c;
   }
}



/*******************************************************/
/* This procedure receives a character string and a    */
/* starting location on the stack (st).  It retrieves the*/
/* file name from the stack and returns it in the      */
/* character string for use by the kernel.             */
/* This procedure was added for debugging purposes.    */
/*******************************************************/
getfilename(file, start)
char file[13];
integer start;
{ integer x=0;
char c;

c = st[start];
while ((x < 12) && ((c == '.')
    || ((c >= 'a') && (c <= 'z'))
    || ((c >= 'A') && (c <= 'Z'))
    || ((c >= '0') && (c <= '9'))))
   { file[x++] = c;
   c = st[start + x];
   }
file[x] = '\0';
}

/*******************************************************/
/* This procedure will dump a section of the stack.    */
/* The procedure receives one parameter which is the   */
/* number of stack locations above and below the       */
/* location pointed at by the p 'register' to be dumped.*/
/* This procedure was added for debugging purposes.    */
/*******************************************************/
prog_stack_dump(depth)
integer depth;
{ integer temp, temp1, max_depth = 50;
FILE *fp, *fopen();
```

```
char c;
unsigned d;

  /* verify the depth */
if (depth <= 0)
   depth = 5;

  /* print a header */
fp = fopen("log", "a");
fprintf(fp, "\n\n                      hex         integer
          binary");
temp = p - depth;

  /* dump the stack */
while ((p + depth >= temp) && (temp <= MAXADDR))
   { c = (temp == p) ? 'p' : ' ';
   fprintf(fp, "\n%c st[%5d] ==     %08x%12d     ",
   c, temp, st[temp], st[temp]);
   temp1 = st[temp];
   d = 0x80000000;
   while (d != 0)
      { if (temp1 & d)
         fprintf(fp, "1");
      else
         fprintf(fp, "0");
      d >>= 1;
      }
   temp++;
   }
fprintf(fp, "\n");
fclose(fp);
}

/*****************************************************/
/* This procedure exits from the kernel with a exit    */
/* status that will allow the kernel to reboot.        */
/* This procedure has been added.                      */
/*****************************************************/
reboot()
{
#ifdef UNIXSYSTEM
quitcurses();
#endif
exit(999);
}

/*****************************************************/
/* This procedure exits from the kernel with a exit    */
/* status that will cause complete exit from the       */
/* Edison system.                                      */
/* This procedure has been added.                      */
/*******************************************************/
quit()
{
#ifdef UNIXSYSTEM
quitcurses();
#endif
exit(0);
}


#ifdef UNIXSYSTEM
/*******************************************************/
/* This procedure makes all the initializations        */
/* neccessary for signals.  Signals is used to handle  */
/* different signals recieved, such as ^C and          */
/* arithmatic errors so that curses can be exited from */
/* gracefully, the Edison virtual machine can be       */
/* rebooted or whatever other actions that are needed  */
/* can be taken.                                       */
/* This procedure has been added.                      */
/*******************************************************/
initsignals()
{ signal(SIGQUIT, reboot);
signal(SIGINT, reboot);
signal(SIGBUS, reboot);
signal(SIGSEGV, reboot);
}

/*******************************************************/
/* This procedure makes all the initializations        */
/* neccessary for curses                               */
/* This procedure has been added.                      */
/*******************************************************/
initcurses()
{ LINES = MAXROW;
COLS = MAXCOLUMN;
initscr();
crmode();
noecho();
leaveok(stdscr, FALSE);
scrollok(stdscr, TRUE);
clearok(curscr, FALSE);
refresh();
}

/*******************************************************/
/* This procedure simply calls the curses procedure to */
/* gracefully exit from curses.                        */
/* This procedure has been added.                      */
```

```
/*********************************************************/
quitcurses()
{ endwin();
}

#endif

/*********************************************************/
/* This procedure is used to print the value of a set. */
/* A set is held in 8 locations on the stack (st), so  */
/* the easy way of printing the set value is to call   */
/* the procedure stack_dump.                            */
/* This procedure was added for debugging purposes.    */
/*********************************************************/
printset(setloc)
integer setloc;
{ integer ssave;
ssave = s;
s = setloc + SETLENGTH - 1;
stack_dump(8);
s = ssave;
}
```

APPENDIX C

THE BENCHMARK TESTING PROGRAM

Edison System Prefix

```
const nl = char(10); sp = ' ';
  linelength = 80 "characters";
  namelength = 12 "characters";
  sectorlength = 256 "integers"
set charset (char)
array line [1:linelength] (char)
array name [1:namelength] (char)
array sector [1:sectorlength] (int)
record position (pages, words: int)
enum word (sixteen_bits)
array program [1:12300] (word)
array stream [1:536] (word)

proc prefix(
  progname: name;
  pdp11: bool;
  maxrow, maxcolumn: int;
  proc select(normal: bool);
  proc cursor(row, column: int);
  proc erase;
  proc display(value: char);
  proc assume(condition: bool; text: line);
  proc accept(var value: char);
  proc pause;
  proc print(value: char);
  proc openread(var file: stream; title: name);
  proc more(var file: stream): bool;
  proc read(var file: stream; var value: char);
  proc mark(var file: stream): position;
  proc move(var file: stream; place: position);
  proc endread(var file: stream);
  proc openwrite(var file: stream; title: name);
  proc write(var file: stream; value: char);
  proc endwrite(var file: stream);
  proc create(drive: int; title: name);
  proc delete(drive: int; title: name);
  proc locate(var drive: int; title: name);
  proc rename(drive: int; old, new: name);
  proc protect(drive: int; title: name; value: bool);
  proc readbool(proc read(var c: char); var value: bool);
  proc readint(proc read(var c: char); var value: int);
  proc readname(proc read(var c: char); var value: name);
  proc writebool(proc write(c: char); value: bool);
  proc writeint(proc write(c: char); value, length: int);
  proc writename(proc write(c: char); value: name);
  proc writeline(proc write(c: char); value: line);
  proc readsector(drive, sectorno: int; var value: sector);
  proc writesector(drive, sectorno: int;
    var value: sector);
  proc subset(first, last: char): charset;
  proc load(title: name): program)

"The Edison-PC System: Benchmark Test Program
                    Date
 Copyright (c) 1982 Per Brinch Hansen"



var x : int

begin
   x := 1;
   while x<=1000 do
      writeint(display,x,5);
      display(nl);
      x := x + 1
   end
end
```

APPENDIX D

THE USER'S INSTALLATION GUIDE

The work and time of installing the Edison System onto a new host environment has been greatly reduced with this project. This guide will provide the programmer with the necessary information to achieve a successful installation.

The first step is to copy all of the supporting system files for the Edison System onto the new host environment. These files include the operating system, the compiler, the editor, and a few miscellaneous programs included for program development. It is also necessary to copy the C source code files onto the host environment. The list of files that must be copied onto the new host environment are shown in Table D.1.

EDISON.C	C source code file
CKERNEL.C	C source code file
OPSYSCOD.PLN	The Edison Operating System
EDIT	The Edison System Editor
COMPILE	The Edison Language Compiler
EDISON1	The Edison Compiler (Pass 1)
EDISON2	The Edison Compiler (Pass 2)
EDISON3	The Edison Compiler (Pass 3)
EDISON4	The Edison Compiler (Pass 4)
CUT	Application Development Aid
PASTE	Application Development Aid
PRINT	Text Printing Program
ASC2ED.C	Source Code for Programmer Aid
ED2ASC.C	Source Code for Programmer Aid

Files Necessary for Installation
on New Host Environment
Table D.1

It will be necessary for the new host environment to have a C compiler that is compatible with the C language as defined by Kernighan and Ritchie. The files ED2ASC.C and ASC2ED.C will require no modification for the new environment. It is simply necessary to compile them into executable form. These files will convert Edison text files into ASCII text files and also from ASCII to Edison.

The only modification the file EDISON.C will require is the specification of the drive and path necessary to start the program CKERNEL (which will be compiled from CKERNEL.C). The remaining modifications must be done to the file CKERNEL.C. The modifications are restricted to some declarations at the beginning of the program and seven routines. These routines are listed in Appendix A.

The changes in the declarations at the beginning of program are very simple. The default drive/path definition must be supplied with the DEFAULTDRIVE declaration. During implementation debugging procedures, it may be necessary to change the trace level. By changing the declaration of TRACE, it is possible to have the system provide varying amounts of information during execution.

The first routines to require modification are READX() and WRITEX(). Modification to these routines is only necessary if full screen control is desired in the new host environment. If screen control is desired, then it is necessary to supply the routine calls in READX() and WRITEX().

The next few routines require similar modifications. The routines of COPY_FILE(), RM_FILE(), and PROTECT_FILE() rely upon the host environment to provide the operations necessary for their execution. These routines make calls to the host environment to invoke commands to copy files, erase files, and protect files. Each of these routines must be modified to utilize the correct syntax for the system commands.

The final two routines are the most difficult to modify. Both CHECK_DRIVE() and GET_DISK_CATALOG() access the disk catalog. These routines also submit a system call to the host environment. They request a directory of the disk catalog and they route the response to a temporary file. The CHECK_DRIVE() routines use the information gained from the directory to determine if the selected drive is valid. Every host environment will give an error message if an invalid directory is requested. This routine looks for the error message. The routine GET_DISK_CATALOG is the most complex routine to modify. It is necessary for this routine to translate the directory supplied by the host environment into something the Edison System can understand. This routine uses the temporary file created to hold the directory information from the disk catalog request. It scans each line and locates the file name, size, and protection status. The host environment must be capable of supplying the size of the file in bytes (8 bits) or words (16 bits). This routine must convert size in bytes into the size in words. (Size in bytes is the most common method used in today's host environments.)

For further enhancements, it may be desired to modify the routines CURSOR() and ERASE(). These routines provide greater usability when implementing full screen control. It is necessary to modify these routines to use the screen control routines available with the host environment.

The implementation of the Edison System is very straightforward. The process has been greatly reduced by the completion of this project. It is now easily installed into a new environment with minimal delay. This guide has outlined the steps necessary to follow when implementing the Edison System in a new host environment.

The Development of the Edison System as a Truly Portable Software Development Environment

by

Michael C. Wonderlich

B. S. Kansas State University, 1985

An ABSTRACT of a MASTER'S REPORT

submitted in partial fulfillment of the

requirements of the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1989

The Edison System was developed by Brinch Hansen to establish an environment for the development of application software. His design is built upon the concept of providing a small portable operating system. The Edison System is comprised of the operating system, a programming language (also called Edison), and some assorted support programs. The entire system is contained in a very small amount of code.

The goal of this project is to rewrite the kernel from assembly language into a high level language. This provides a more portable system than before. Since most host environments have compilers for the language C, this is the high level language the kernel has been written in. Now only a few minor changes are necessary in the kernel to move it into a new host environment.

With the success of this project, it is now possible to easily port the Edison System to new host environments. The Edison System was successfully implemented in several versions of Unix and MS-DOS. This report discusses the history leading to the development of this project. A full account of the project is included for anyone wishing to further this work. Hopefully the work accomplished in this project will be used to further the development of a truly portable system environment. This system will provide a common environment to develop application software for all uses in the computer industry.