The City University

B. Sc. Degree in Computer Science

Project Report

"An operating system for a minicomputer"

by

Nigel Eke

This report is an account of a project carried out in the final year of the course and constitutes part of the final examination in Computer Science.

Date submitted:

"21-May-79"

The City University

B. Sc. Degree in Computer Science

Project Report

"An operating system for a minicomputer"

by

Nigel Eke

This report is an account of a project carried out in the final year of the course and constitutes part of the final examination in Computer Science.

Date submitted:

"21-May-79"

Project report

Abstract

This report describes an operating system for a minicomputer. It covers operating system fundamentals and is followed by an account of the design and implementation of a small system based on a PDP-11 minicomputer.

Nigel Eke

21-May-79

CONTENTS

PAGE

Section 1 1

Operating Systems 1

1. A brief history 1

2. Modern operating systems 1
2.1 Single user systems 1
2.2 A multiprogramming system 2
2.3 Timesharing systems 3
2.4 Other systems 4
2.4.1 Real time 4
2.4.2 Dedicated systems 4

3. Present day commercial trends 4
3.1 The amateur market 4
3.2 The business market 4
3.3 The neglected area 5

Section 2 6

The design of a single user system 6

1. Principal objectives 6

2. Software specifications 6
2.1 The command device 6
2.1.1 Command device use 6
2.1.2 User requirements 7
2.1.2.1 Direct output 7
2.1.2.2 Controlled output 7
2.1.2.3 Text output 7
2.1.2.4 Cursor control 7
2.1.2.5 Character input 8
2.1.2.6 Echo control 8
2.1.2.7 Program escape 8
2.2 File handling 8
2.2.1 File description 8
2.2.2 File protection 9
2.2.3 User requirements 9
2.2.3.1 Looking for data on a file 9
2.2.3.2 Opening files 9
2.2.3.3 Reading and writing 9
2.2.3.4 Closing a file 10
2.2.3.5 Deleting a file 10
2.2.3.6 Renaming a file 10
2.2.3.7 Indexing 10
2.3 Simplicity of use 10
2.3.1 Handling the system 10
2.3.2 Writing programs for the system 11
2.4 Error handling 11

2.4.1 System errors 11
2.4.1.1 Fatal errors 12
2.4.1.2 Non-fatal errors 12
2.4.2 User errors 12
2.5 Other facilities 13
2.5.1 Delay 13
2.5.2 Input until end of line 13
2.5.3 Input file name 13
2.5.4 Run file 13

3. Suggested implementation methods 13
3.1 Command device control 13
3.2 Disc structure 14
3.2.1 File directory 14
3.2.2 Bitmap 14
3.2.3 Disc allocation 14
3.3 Disc control 14

4. Design of user routines 14
4.1 Output to display 15
4.1.1 Direct output 15
4.1.2 Controlled output 15
4.1.3 Text output 16
4.1.4 Cursor control 16
4.2 Keyboard input 16
4.2.1 Test character 16
4.2.2 Input character 17
4.2.3 Test line 17
4.2.4 Input line 18
4.2.5 Echo control 18
4.3 File handling 19
4.3.1 Look 19
4.3.2 Open for input 19
4.3.3 Open for output 20
4.3.4 Read 20
4.3.5 Write 21
4.3.6 Close 21
4.3.7 Deleting 22
4.3.8 Renaming 22
4.3.9 Indexing 23
4.4 Error handling 23
4.4.1 Fatal errors 23
4.4.2 Warnings 23
4.4.3 Set warning inhibit 23
4.4.4 Set escape inhibit 24
4.5 Other executive routines 24
4.5.1 Delay 24
4.5.2 Input until end of line 24
4.5.3 Input file name 24
4.5.4 Run file 25
4.5.5 Exit 25
4.6 Device control routines 26
4.6.1 Display interrupt service 26
4.6.2 Keyboard interrupt service 26
4.6.3 Disc control 26

Section 3 27

The system implementation 27

1. SUSIE 27

2. Hardware 27

3. Software 27
3.1 Command terminal i/o 27
3.1.1 Output 27
3.1.2 Input 28
3.2 Disc file i/o 29
3.2.1 Basic input and output 29
3.2.2 General tables 29
3.2.2.1 Disc allocation 29
3.2.2.2 Bitmap 30
3.2.2.3 File directory 30
3.2.2.4 File state table 31
3.2.3 File structure 31
3.2.4 Individual file control 32
3.2.4.1 Local file descriptors 32
3.2.4.2 The i/o buffer 33
3.2.4.3 The open file chain 33
3.3 Error handling 34
3.3.1 Non-fatal errors (warnings) 34
3.3.2 Fatal errors 34
3.3.2.1 System errors 35
3.3.2.2 Machine errors 35
3.4 Command decoding 35
3.5 Other programs 35
3.5.1 Initialization 35
3.5.1.1 Disc initialization 35
3.5.1.2 File initialization 35
3.5.2 Bootstrap 36
3.5.3 Utility programs 36
3.5.4 Test programs 36

4. Assembling 36

Section 4 37

Using SUSIE 37

1. Initialization of a disc 37

2. Running programs on SUSIE 37

3. Preparing programs for SUSIE 38
3.1 The DOS-BATCH system 38
3.2 SUSIE system macros 38
3.3 User source programs 38
3.4 Assembling the source program 38
3.5 Placing programs onto SUSIE 38

4. System macros 39

4.1 Name 39
4.2 Mess and Lmess 39
4.3 Lfd and Lfdb 39
4.4 Exit 40
4.5 Delay 40
4.6 Cpos 40
4.7 Ochar 40
4.8 Dchar 40
4.9 Dtext 40
4.10 Swrni 41
4.11 Warn 41
4.12 Error 41
4.13 Tchar 41
4.14 Inchr 41
4.15 Tline 42
4.16 Inlin 42
4.17 Ineol 42
4.18 Infnm 42
4.19 Steco 42
4.20 Sesci 43
4.21 Look 43
4.22 Openi and Openo 43
4.23 Read 43
4.24 Write 43
4.25 Close 44
4.26 Breof and Brneof 44
4.27 Delete 44
4.28 Rename 44
4.29 Index 44
4.30 Run 45

5. File utility program (FIDO) 45
5.1 General description 45
5.2 Commands 45
5.2.1 File transfer 45
5.2.2 Renaming a file 45
5.2.3 Deleting a file 46
5.2.4 Running a file 46
5.2.5 File indexing 46
5.3 Command concatenation 46

Section 5 47

Project summary 47

1. Objectives 47

2. Achievements 47
2.1 Simplicity of use 47
2.2 Memory size 47
2.3 Error messages and error handling 47
2.4 Dealing with the 'idiot' user 48

3. Future work for the project 48
3.1 Memory size 48
3.2 Contiguous files 48

3.3 Interpreters 49
3.4 Multiprogrammable systems 49

Section 6 51

Appendices 51

1. References 51

2. Acknowledgments 51

3. Computer printout 51

Project report

Section 1

Operating Systems

Nigel Eke

21-May-79

Section 1

Operating Systems

1. A brief history

Operating systems were unknown in the first generation computers. The programs running on these computers were stand-alone. That is each individual program controlled the TOTAL operations of the computer. The execution of a program would only take about a second but would require five minutes, or more, of human intervention. When errors occurred the time required to locate and correct them would be much greater.

It was because of this that the operating system was introduced. This, initially, took the form of a program that regained control of the machine when an error was committed by the users program. It allowed various registers and locations to be examined at computer, rather than human, speeds.

Operating systems then began to aid the user, and the users program, by controlling the input and output functions and permitting control to pass from one program to the next.

2. Modern operating systems

2.1 Single user systems

A single user operating system allows the user to make requests, usually through system commands, about the execution of various programs. The following list gives possible examples:-

LOAD-	Load a program into memory
RUN-	Run program already in memory
HALT-	Temporarily halt the program
CONT-	Continue execution of a program
RESTART-	Restart execution of the loaded program

The system could also permit various input and output channels to be allocated to one of a number of peripherals. E.g.:-

SET 1, PR-
Set channel 1 to paper tape reader

SET 2, DF, OUTDAT-
Set channel 2 to disc file called OUTDAT

Using the above commands a user could load a program, set output to a file, input from paper tape reader and begin execution of the program. During the execution of the program data from the paper tape reader may become exhausted. The user could then temporarily halt the program, reset input to a keyboard or another disc file, and continue the program until completion.

Other commands could also be available to aid error detection. These, for example, may be:-

EXAM address, number of words
DEPOS location, new contents

All facilities of the computer are available to the user at any time.

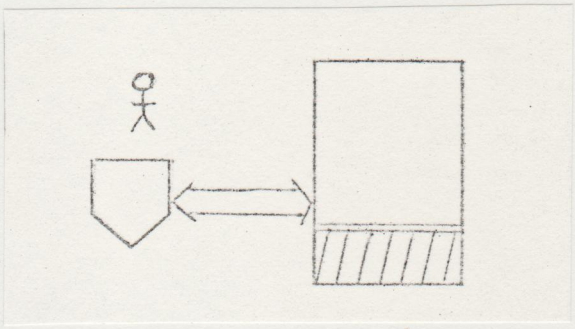

Fig 2.1 A single user system

2.2 A multiprogramming system

When a computer system is dealing with a large quantity of input and output much central processor time is wasted. This is because the entire system is waiting for data to be transfered to or from the relatively slow peripheral. The computer could be running other programs at these stages. A multiprogramming system gets around this problem by having a number of programs in memory at the same time. When one program begins an input or output operation a second program will be allowed central processing time. When this program initiates an input or output operation a third program will be allowed central processing time. The process continues in this fashion until all programs have finished. Again, commands may be issued, from a central terminal, which will control the execution of the programs and the allocation of peripherals.

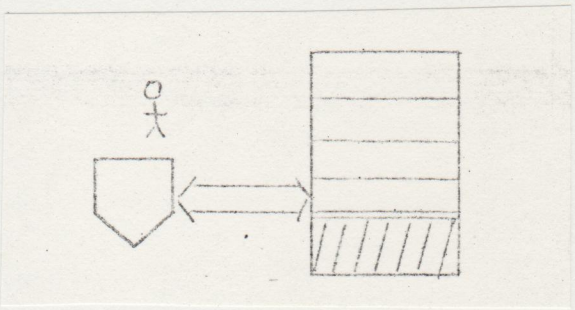

Fig 2.2 A multiprogramming system

2.3 Timesharing systems

This is similar, in many respects, to a multiprogramming system. The main difference is that it allows a number of users to interact with the computer, apparently simultaneously. Each user appears to have the computer dedicated to his particular problem. It is the operating systems responsibility to schedule each program for a predetermined time. (This is generally called the timeslice and lasts about 1/100 th - 1/10 th second). It is also the duty of the operating system to safeguard each users allocated memory and supervise user messages and assign peripherals.

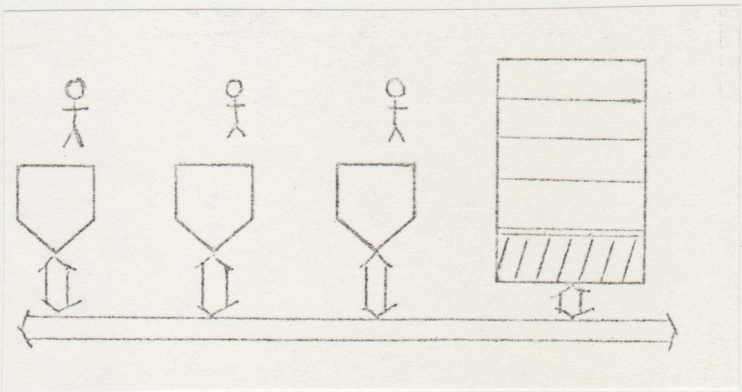

Fig 2.3 A timesharing system

2.4 Other systems

2.4.1 Real time

Real time refers to a system in which the processing of data input to the system occurs virtually simultaneously with the event generating the input. The data usually has to be processed quickly to enable the results to be used as feedback information.

2.4.2 Dedicated systems

Certain systems are used specifically for one purpose and the operating systems have to be designed to accomplish the necessary task. An example of this kind of system would be the control of a steel mill. With the advent of microprocessors the dedicated system will become more widespread. (Current examples include TV games, intelligent terminals and fuel injection control for cars).

3. Present day commercial trends

3.1 The amateur market

With the onset of the 'microprocessor revolution' a number of companies have produced microprocessor computing systems for the amateur. These have been aimed towards the people in the hobby electronics field. The operating systems for these hobbyists enable very crude access to the machine by commands such as:-

EXAMINE	a memory location
DEPOSIT	to a memory location
RUN	from a certain address

Memory dump facilities are also available to allow coarse and somewhat clumsy debugging.

Having studied listings of some of the systems the author feels that they are:-

1) Difficult and tedious to use
2) Inefficiently written

Much can be done to improve this situation.

3.2 The business market

The 'microprocessor revolution' has also allowed small companies to control their business with the use of computers. Until recently only the larger companies could afford to run

and maintain the computers and air conditioning systems they require. Now the smaller companies may purchase a computer and control stock, employees pay etc... No special air conditioning systems are needed and the computer may run from the domestic power supply.

Other business uses include word processing, work scheduling (such as critical path analysis).

These systems have been produced for specific purposes and often include professional 'extras' such as consultation and maintenance contracts. Both these points make these systems still relatively expensive for the amateur computer hobbyist.

3.3 The neglected area

The microprocessor market bears many similarities to that of the music industry. The following table illustrates the analogy between the two markets:-

£	MUSIC	COMPUTING
10^1	POCKET RADIOS	POCKET CALCULATORS
10^2	MID-FI	PERSONAL COMPUTERS
10^3	HI-FI	?
10^4	PROFESSIONAL SOUND SYSTEMS	BUSINESS COMPUTER SYSTEMS

Table 3.1 Music - Computer comparison

It can be seen from the above table that amateur computing is sadly lacking a relatively cheap high performance system.

Project report

Section 2

The design of a single user system

Nigel Eke

21-May-79

Section 2

The design of a single user system

1. Principal objectives

The main objective of this project is to write a single-user system to fill in the gap indicated in section 1-3.3.

The proposed hardware system should include:-

central processor	A high performance microprocessor
keyboard/display	The system control terminal
floppy disc unit	A high speed backing store (C.f.-Cassette)
Memory/buffers etc	Around 8k-12k minimum

The above should provide an integrated but easily expandable system. It fulfils the price range required for a Hi-Fi personal computer and includes all the basic, but necessary, equipment to allow programs to be run and retained. The major difference from the usual personal computers is the inclusion of the floppy disc unit. A cassette tape recorder is normally used for cheap backing store. The floppy disc allows much quicker access to stored files and does not need the manual interjections that are required by cassettes. (I.e. forward wind etc).

2. Software specifications

Having discussed the hardware required it must be stated that no computer is complete without some software to drive it.

The following table shows what objectives the operating system should fulfil:-

1) Control of input and output to command device
2) Easy access to files stored on disc
3) Simple to use
4) Easily understood error messages

2.1 The command device

2.1.1 Command device use

Many systems attempt to be as general as possible. This allows easy modifications for more specialised user requirements. Making the operating system device independant greatly increases the complexity and therefore the amount of memory essential.

Having a separate command device input/output routine simplifies the operating system and allows greater control of the system should parts of the system deteriorate and become inoperative.

Many of the current systems use a visual display unit as the main input/output terminal. This is easily the most convenient device to use for amateur systems, especially as it requires no paper (which all adds to the expense of the system).

2.1.2 User requirements

2.1.2.1 Direct output

The user must be allowed to output characters directly to the display. This enables special display functions (such as cursor control, clear screen) to be used if provided.

2.1.2.2 Controlled output

Certain characters provide special effects on a terminal. For example <TAB> goes to the next column on the terminal. Most terminals do not provide this as a direct output character and so <TAB> should be replaced by the correct number of spaces.

Other characters (most of the control characters) produce no effect on the terminal and should be replaced by their name. This means their existance can be verified. For example:- Control-A is output as '<SOH>' to show start of heading.

2.1.2.3 Text output

Very often the user will wish to output text to the screen. This is easily achieved by sending each character of the text through the controlled output.

2.1.2.4 Cursor control

If the user is to take full advantage of the cursor control allowed by the direct output it is necessary to make available the current cursor position. When direct outputs are made the new position must be specified:-

direct output (line feed, cursor position)

direct output (backspace, cursor position-1)
direct output (carriage return, 0)

2.1.2.5 Character input

Four routines are necessary or useful for reading characters from the terminal keyboard:-

1) Test to see if character is available. Return character if it is, otherwise return NULL
2) Read a characater - wait until character is available
3) Test to see if complete line has been input. Return next character in line if so, otherwise return NULL
4) Wait for complete line to be input and then return next character in line.

The line input routine allows rubout and delete processing to be completed before any characters are sent to the users program.

E.g.:-
ABD<rubout>CDE<end of line>
will send ABCDE<end of line> to the program.

2.1.2.6 Echo control

It is helpful if all characters that are input using the above routines are echoed on to the display screen. But, again to allow the user full control of the terminal, it should be possible to turn the echo facility on and off.

2.1.2.7 Program escape

It is possible that user programs may get stuck in loops and some means of escape is necessary to get back to the operating system. A control key '<escape>' should allow this and, like the echo, it should be turned on and off as the user requests.

2.2 File handling

2.2.1 File description

Each file should be identified by an alphanumeric file name.

To simplify the operating system no distinction should be made between file types. This means binary files, text files and special data files will all be treated alike.

2.2.2 File protection

Many systems provide very sophisticated protection. They provide protection against deletion, reading, overwriting and other users. All this enlarges the size of the operating system.

Because this system is aimed at the personal computer user, who is possibly a beginner at computing, it is felt that some degree of file protection is necessary.

The most elegant method of providing protection, in the authors opinion, is to provide backup copies of old files. The easiest method of providing backups is just to keep one old copy as opposed to multiple copies.

Some file accesses (e.g. open for writing) should create a backup from the current main file. Whereas other file accesses (e.g. delete) should create a new main file from the last backup.

2.2.3 User requirements

2.2.3.1 Looking for data on a file

Some method of gaining data on the state of various files must be available. This is necessary for other file operations (to be described later) and may be of use to the systems user.

Data returned should consist of:-

1) Does file exist ?
2) Does it have a backup ?
3) Is file open ?
4) If (3) is it open for reading or writing ?

2.2.3.2 Opening files

Before reading or writing files various pointers and general states must be initialized. This should be done by routines that open a file for reading or for writing.

Checks must be made to see if the file is already open or not.

2.2.3.3 Reading and writing

These routines should read or write the next byte (character) from/to a file and set end of file flags if

necessary.

Files must be opened before reading or writing commences.

2.2.3.4 Closing a file

This should reset and clear the flags that have been set by the open routines. It should then allow access to the file again.

2.2.3.5 Deleting a file

Deleting files is necessary to prevent the available disc space reaching zero. The choice of the files to be deleted should be left to the user and not to the system itself.

2.2.3.6 Renaming a file

Renaming can be done by reading from a file and writing to the new file. This would require an enormous number of disc accesses which is not a good idea for floppy disc storage. To cut down the amount of disc access a special routine is required.

2.2.3.7 Indexing

Some method of discovering what programs reside on disc must be available.

2.3 Simplicity of use

This covers two areas:-
1) Handling the system
2) Writing programs for the system

2.3.1 Handling the system

In most operating systems two commands occur more frequently than any others. These are LOAD and RUN. Commands such as COMPILE or EDIT can be reduced to LOAD COMPILER, RUN or LOAD EDITOR, RUN.

Under this system the combination of commands LOAD and RUN should be taken by default and the user need only enter the name of the program to be run. This means each program available to the user constitutes a system command.

```
#PNAME=ABCD<end of line>

                    <end of line> means execute command
       ABCD<end of line> are characters still available to
       the user
      = delimiter after program name (any non-alphanumeric)
     name of program to be run
# prompt output by system
```

An example of a program creation sequence might be

```
#INPUT=PROG<end of line>
     lines of data for input
#COMP=PROG<end of line>
     compile error messages
#EDIT=PROG<end of line>
     edit commands
#COMP=PROG<end of line>
     successful compilation
#BPROG=program data <end of line>
     run of binary program created by compiler.
```

2.3.2 Writing programs for the system

The user should have available a number of system macros to simplify the creation of the programs. (These are described in more detail in section 4-3.2).

2.4 Error handling

2.4.1 System errors

When errors occur in many systems the user is faced with a barrage of numbers or a cryptic message. Because the system is aimed at a possible first time user in computing the error message that occurs must say fairly clearly what the error is. For example, if a file is to be opened but it does not exist, the message "!! OPEN- No file" states what the error is and where it occured.

System errors fall into two categories:-

1) Fatal errors
2) Non-fatal errors (or warnings)

2.4.1.1 Fatal errors

If a fatal error occurs a message, preceeded by '!!', should be output indicating what the error is.

Example:-

```
!! DELETE- File open
!! OPEN- No file
```

There is no way a program can continue after executing a fatal error so all files are closed and the program terminated.

2.4.1.2 Non-fatal errors

Some errors that occur are not fatal, although they generally show that something has gone wrong with the program and it will produce fatal errors at a later stage.

If a non-fatal error occurs a message, preceeded by '??', should be printed.

Example:-

```
?? CLOSE- File closed
?? DELETE- No file
```

The program can continue after these warning messages have been output.

2.4.2 User errors

The user may detect errors, possibly on the data being input to his program, and may wish to take certain action when detected. Two routines available to the user should be:-

1) Fatal errors-output message and force termination
2) Warning- output message and return to program

It must also be possible to turn the warning messages off if not wanted.

E.g. The messages:-

```
?? Overflow
?? SQRT of negative number
?? Log zero
```

are sometimes not required when default values have been supplied.

2.5 Other facilities

Some routines can be made available to the user at very little extra cost.

2.5.1 Delay

This should allow a program to be delayed for a period defined by the user.

2.5.2 Input until end of line

This should allow characters to be input from the keyboard until '<end of line>' is read.

2.5.3 Input file name

This should allow a file name to be input from the display keyboard and stored in a file description block (described in section 3-3.2.4.1).

The first six alphanumeric characters should make up the name.

E.g. ABC = ABC<sp><sp><sp>
ABCDEFGH = ABCDEF

An illegal name (first character input is non-alphanumeric) should leave the file description block untouched.

2.5.4 Run file

This should allow a file (represented in a file description block) to be loaded into main memory and run.

3. Suggested implementation methods

3.1 Command device control

The input and output for the command terminal is best controlled by interrupt handling. The main advantages of this are:-

1) On output other central processor operations may continue while messages are being sent to

the terminal.

2) On input the user may enter a number of lines of data ahead of the current process.

The main disadvantages are:-

1) Buffers are required for input and output therefore using extra main memory.
2) Error handling is required when the buffers become full - again using extra memory.

3.2 Disc structure

3.2.1 File directory

The file directory describes a file that resides on disc. It holds the file name and pointers to a chain of disc blocks.

3.2.2 Bitmap

The bitmap holds data on the free blocks on disc. Each bit in the bitmap table is set if the appropriate block on the disc is free.

3.2.3 Disc allocation

The disc must hold:-

1) Bitmap
2) File directory
3) Operating system
4) User files

3.3 Disc control

To avoid further complication of the operating system it is thought best that disc control is direct. I.e.- it is not controlled by interrupts. After each disc access the central processor should wait until the access is complete.

4. Design of user routines

All routines for the operating system have been designed using a pseudo high level language. The following paragraphs give listings of these routines and are linked directly to the subroutines in the listing given in section 6.

4.1 Output to display

4.1.1 Direct output

```
PROC outchar (BYTE char;INT new column count);
BEGIN
   WHILE output buffer = full DO
   OD;
   place char in o/p buffer;
   column count:=new column count;
END of outchar;
```

4.1.2 Controlled output

```
PROC display char (BYTE char);
BEGIN
   IF char<>control AND char<>delete THEN
      outchar (char, cursor position+1);
   ELSE
   IF char=<end of line> THEN
      outchar (<cr>, 0);
      outchar (<lf>, 0);
   ELSE
   IF char = <tab> THEN
      INT number of spaces;
      number of spaces:=8-(cursor postion MOD 8);
      FOR number of spaces:=number of spaces-1
       WHILE number of spaces>=0 DO
         outchar (<sp>, cursor position+1);
      OD;
   ELSE
   IF char = <bell> THEN
      outchar (<bell>, cursor position);
   ELSE
      outchar ('<', cursor position+1);
      FOR each character in name of control char DO
         outchar (name character, cursor position+1);
      OD;
      outchar ('>', cursor position+1);
   FI;
END of display char;
```

4.1.3 Text output

```
PROC display text (STRING message);
BEGIN
   BYTE char;
   FOR char:=each character in message DO
      display char (char);
   OD;
END of display text;
```

4.1.4 Cursor control

```
INT PROC cursor position;
BEGIN
   column count;
END of cursor position;
```

4.2 Keyboard input

4.2.1 Test character

```
BYTE PROC test inchar;
BEGIN
   BYTE char;
   IF char in processed buffer THEN
      char:=next char in processed buffer;
   ELSE
   IF char in i/p buffer THEN
      IF echo flag = true THEN
         display char (char in i/p buffer);
      FI;
      char:=next char in i/p buffer;
   ELSE
      char:=<null>;
   FI;
   IF char = <end of line> THEN
      number of lines input:=number of lines input-1;
   FI;
   char;
END of test inchar;
```

4.2.2 Input character

```
BYTE PROC inchar;
BEGIN
   BYTE char;
   FOR char:=test inchar WHILE char = <null> DO
   OD;
   char;
END of inchar;
```

4.2.3 Test line

```
BYTE PROC test inline;
BEGIN
   BYTE char;
   IF number of lines input >= 1 THEN
      IF processed buffer = empty THEN
         process line until <end of line> found;
      FI;
      test inchar;
   ELSE
      <null>;
   FI;
END of test inline;
```

4.2.4 Input line

```
BYTE PROC inline;
BEGIN
   IF number of lines input >= 1 THEN
      IF processed buffer = empty THEN
         process line until <end of line> found;
      FI;
      test inchar;
   ELSE
      WHILE number of lines input = 0 DO
         process line until end of processed buffer;
      OD;
      process line until <end of line> found;
      test inchar;
   FI;
END of inline;
```

4.2.5 Echo control

```
PROC set echo (BIT flag);
BEGIN
   echo flag:=flag;
END of set echo;
```

4.3 File handling

4.3.1 Look

```
PROC look (LFD file);
BEGIN
   get hash number of file;
   IF file exists THEN
      IF file open THEN
         IF open for i/p THEN
            state open for i/p;
         ELSE
            state open for o/p;
         FI;
      ELSE
         state file closed;
      FI;
      IF backup exists THEN
         state backup exists;
      ELSE
         state backup does not exist;
      FI;
      state file exists;
   ELSE
      state file does not exist;
   FI;
END of look;
```

4.3.2 Open for input

```
PROC openi (LFD file);
BEGIN
   look (file);
   IF file does not exist THEN
      error ("!! openi- no file");
   ELSE
   IF file open for o/p
    OR open for i/p 127 times THEN
      error ("!! openi- file open");
   ELSE
      attach to open file chain;
      state file opened in fst; (file state table)
      state open for i/p in lfd;
      read first block into buffer;
      set pointer to buffer;
      IF number of legal bytes > 0 THEN
         clear end of file found;
      ELSE
         set end of file found;
      FI;
   FI;
END of openi;
```

4.3.3 Open for output

```
PROC openo (LFD file);
BEGIN
   look (file);
   IF file exists THEN
      IF file open THEN
         error ("!! openo- file open");
      ELSE
      IF file has backup THEN
         delete block chain (from first backup block);
      FI;
   FI;
   IF allocate block = unsuccessful THEN
      error ("!! openo- disc full");
   ELSE
      make current file into backup version;
   FI;
   attach to open file chain;
   indicate file open in fst;
   indicate open for writing in lfd;
   set block number as allocated block;
   reset byte pointer to start of buffer;
END of openo;
```

4.3.4 Read

```
BYTE PROC read (LFD file);
BEGIN
   IF file not open for i/p by lfd THEN
      error ("!! read- file not open");
   ELSE
   IF end of file on previous read THEN
      error ("!! read- end of file");
   ELSE
      read next byte from buffer;
      IF pointer beyond legal buffer limit THEN
         IF next block = null THEN
            set end of file found;
         ELSE
            read next block into buffer;
            reset byte pointer;
            IF number of legal bytes = 0 THEN
               set end of file found;
            FI;
         FI;
      FI;
   FI;
END of read;
```

4.3.5 Write

```
PROC write (LFD file; BYTE byte);
BEGIN
   IF file not opened for o/p by lfd THEN
      error ("!! write- file not open");
   ELSE
      put byte into buffer;
      IF at end of buffer THEN
         get current block number;
         next block number:= allocate new block;
         write buffer to current block;
         IF no new block exists THEN
            error ("!! write- full disc");
         ELSE
            reset buffer pointer;
         FI;
      FI;
   FI;
END of write;
```

4.3.6 Close

```
PROC close (LFD file);
BEGIN
   IF file not opened by lfd THEN
      warning ("?? close- file closed");
   ELSE
      IF file open for o/p THEN
         get current block number;
         set next block pointer to <null>;
         write block to current block number;
      FI;
      detach file from open file chain;
      state file closed in fst;
      state file closed in lfd;
   FI;
END of close;
```

4.3.7 Deleting

```
PROC delete (LFD file);
BEGIN
   look (file);
   IF file does not exist THEN
      warning ("?? delete- no file");
   ELSE
   IF file is open THEN
         error ("!! delete- file open");
   ELSE
      reclaim blocks taken by main file;
      IF backup exists THEN
         make backup into main file;
         eliminate the backup;
      ELSE
         eliminate file from directory;
      FI;
   FI;
END of delete;
```

4.3.8 Renaming

```
PROC rename (LFD old file, new file);
BEGIN
   look (old file);
   look (new file);
   IF old file does not exist THEN
      error ("!! rename- no file");
   ELSE
   IF old file open OR new file open THEN
      error ("!! rename- file open");
   ELSE
      IF new file exists THEN
         IF new file has backup THEN
            delete chain of backup file;
         FI;
         make current main file into backup;
      FI;
      make new file copy of old file;
      IF old file has backup THEN
         make main old into copy of the backup;
         eliminate the backup;
      FI;
   FI;
END of rename;
```

4.3.9 Indexing

```
PROC index (INT file number);
BEGIN
   return name of file at directory position;
   return exist state;
   return backup exist state;
   return open state;
END of index;
```

4.4 Error handling

4.4.1 Fatal errors

```
PROC error (STRING message);
BEGIN
   display char (<end of line>);
   display text (message);
   display char (<end of line>);
   perform exit;
END of error;
```

4.4.2 Warnings

```
PROC warning (STRING message);
BEGIN
   IF warning inhibit = false THEN
      display char (<end of line>);
      display text (message);
      display char (<end of line>);
   FI;
END of warning;
```

4.4.3 Set warning inhibit

```
PROC set warning inhibit (BIT flag);
BEGIN
   warning inhibit flag:=flag;
END of set warning inhibit;
```

4.4.4 Set escape inhibit

```
PROC set escape inhibit (BIT flag);
BEGIN
   escape inhibit flag:=flag;
END of set escape inhibit;
```

4.5 Other executive routines

4.5.1 Delay

```
PROC delay (INT clock ticks);
BEGIN
   set up clock;
   wait for zero count;
END of delay;
```

4.5.2 Input until end of line

```
PROC input until end of line (BYTE last char);
BEGIN
   WHILE last char <> <end of line> DO
      last char:=inline;
   OD;
END of input until end of line;
```

4.5.3 Input file name

```
PROC input file name (LFD file);
BEGIN
   look (file);
   IF file open THEN
      error ("!! infnm- file open");
   ELSE
      read file name into lfd (using inline);
   FI;
END of input file name;
```

4.5.4 Run file

```
PROC run (LFD file);
BEGIN
   close all files;
   IF load (file) = cannot THEN
      error ("!! run- load error");
   ELSE
      run file;
   FI;
END of run file;
```

4.5.5 Exit

```
PROC exit;
BEGIN
   close all files;
   reset stack pointers;
   re-enable keyboard interrupts;
   return to command decoder;
END of exit;
```

4.6 Device control routines

4.6.1 Display interrupt service

```
IF o/p buffer <> empty THEN
   display next character in o/p buffer;
FI;
return from interrupt;
```

4.6.2 Keyboard interrupt service

```
IF char = <ESC> AND esc inhibit = clear THEN
   empty i/p buffer;
   empty o/p buffer;
   number of lines input := 0;
   error ("<< escape >>");
ELSE
   IF keyboard buffer full THEN
      warning ("?? keyboard buffer full");
   ELSE
      IF char = <end of line> THEN
         number of lines input :=
               number of lines input + 1;
      FI;
      place character in i/p buffer;
   FI:
FI;
return from interrupt;
```

4.6.3 Disc control

```
set number of words (256);
set direction (r/w);
set core address;
set up sector number;
perform transfer;
wait for transfer to finish;
return;
```

Section 3

The system implementation

Nigel Eke

21-May-79

Section 3

The system implementation

1. SUSIE

This section covers the implementation of the single user operating system described in section 2. SUSIE is an acronym for Single USer Interactive Executive and is the name given to the operating system.

2. Hardware

SUSIE has been implemented on a PDP 11/34. This represents the high performance microprocessor described in section 2-1. Since no floppy disc is attached to the 11/34 its cartridge disc has been used instead. Only 1 megabyte of the cartridge disc has been used to obtain a better simulation.

3. Software

This can be divided into five main categories:-

1) Command terminal i/o
2) Disc file i/o
3) Error handling
4) Command decoder
5) Other programs

3.1 Command terminal i/o

The suggested implementation methods are used for the command terminal. (I.e. interrupt handling).

3.1.1 Output

An output buffer is set aside for the display. A character is placed into this buffer at each output request and the interrupt bit for the display is enabled. If the buffer is full when output is requested then the processor will wait until the characters at the beginning of the buffer have been output.

When an 'output' interrupt occurs a character is taken from the buffer.

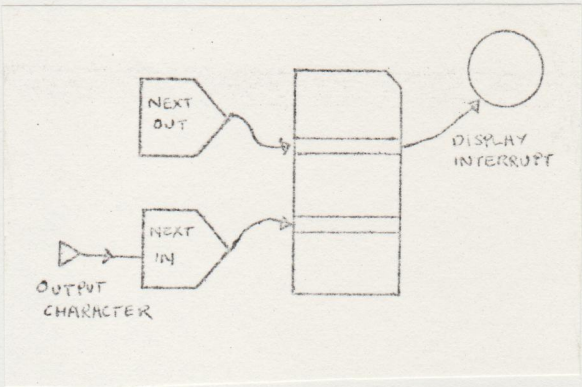

Fig 3.1 Output buffer structure

3.1.2 Input

The input structure is slightly more complicated than the output described above.

The first stage is similar in that:-

1) A character is entered in the buffer at each interrupt. ALL characters typed at the keyboard are put in this buffer.
2) A character is removed from the buffer, and echoed if required, at each 'input' request.

The second stage allows the 'line input' processing to be carried through.

This takes characters from the keyboard buffer and places them in the processed buffer. Each 'delete' removes the previous character from the buffer and the screen. Each 'cancel' erases the last complete line from the buffer and screen.

This continues until an 'end of line' character is obtained.

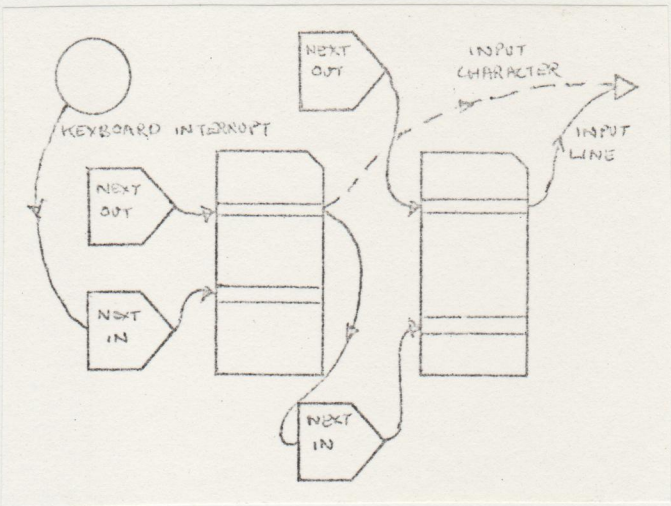

Fig 3.2 Input buffer structure

The user is allowed two types of control over the input:-

1) Echo- Each input character may or may not be echoed automatically onto the screen.
2) Escape- detection of 'escape' allows exit from the user program. This may be inhibited.

3.2 Disc file i/o

3.2.1 Basic input and output

ALL the direct input and output is controlled through one procedure. This is the only part of the program that needs to be changed when the system is implemented on a floppy disc.

3.2.2 General tables

3.2.2.1 Disc allocation

The disc is divided into a number of areas:-

1) Bitmap
2) File directory
3) SUSIE
4) File blocks

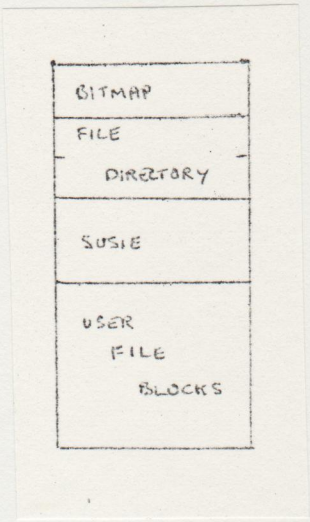

Fig 3.3 Disc allocation map

3.2.2.2 Bitmap

The bitmap is used to indicate which blocks on the disc have been allocated to the system or to user files.

The number of bits required for this bitmap is given by:-

$$\frac{\text{disc size (bytes)}}{\text{bytes/block}} = \frac{\text{1 Mbyte}}{512}$$

$$= (2^{20})/(2^{9}) = (2^{11})$$

Since there are 8 bits/byte the bitmap size is 256 bytes. Each bit in the bitmap is set if the block is free.

The bitmap is held in main memory during system operation. It is retained in the first half of sector zero on the disc and is rewritten at each program exit.

3.2.2.3 File directory

The file directory contains the filenames and pointers to the first blocks of the files. Each file name (6 alphanumeric characters) is stored in two words and one word is taken up for the initial block pointer of the main file and the backup file. A maximum of 128 files are allowed.

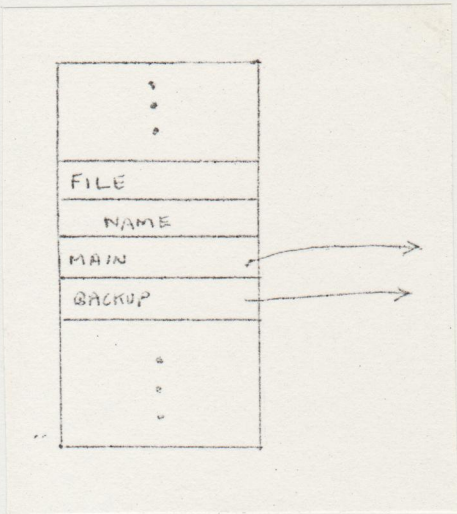

Fig 3.4 File directory

Like the bitmap this table is stored in main memory during system operation and is rewritten to disc (sectors 1 and 2) at each program exit.

3.2.2.4 File state table

Some means of recording the current state of any one particular file must exist because the user must not be allowed to commit errors such as:-

1) Opening file for o/p that is already open
2) Deleting an open file

This is done in the file state table (fst). It has one entry for each possible file (128 entries in all). Each entry (one byte) can take the following values:-

0 - file closed
-1 - file open for output
+n - file open for input (n times)
1<=n<128

3.2.3 File structure

User files are stored on disc as a series of chained blocks. The initial block is pointed to by the file directory entry. Each subsequent block is pointed to by the last word of the preceeding block.

The number of legal bytes of data held by each block is stored in the first word of that block.

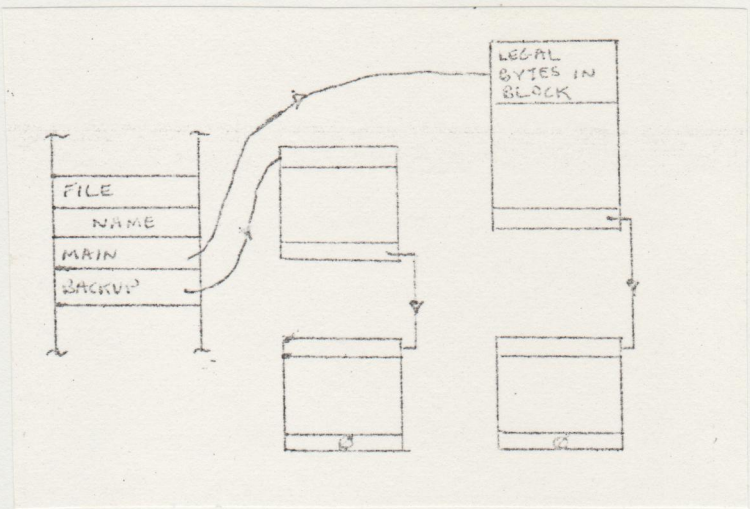

Fig 3.5 File structure

3.2.4 Individual file control

3.2.4.1 Local file descriptors

A local file descriptor (lfd) is set up in the users program to control file accesses. An lfd needs to be set up for each type of manipulation required for a file. E.g.- If ONE file needs to be read and re-read character by character then it can be opened for input twice and this will require TWO serepate lfds.

The information held in an lfd is:-

file name	6 alphanumeric radix 50 characters
next lfd	makes up 'open file chain' (see 3.2.4.3)
last lfd	also part of 'open file chain'
open closed	0 - closed -1 - open for output +n - open for input
end of file	if open for input then- 0 - End of file not yet found 1 - End of file found on previous read
file number	holds the number of the entry of this file in the file directory and the file state table.
i/o buffer	points to the i/o buffer (see 3.2.4.2)
current char	points to next byte in buffer to be read or written

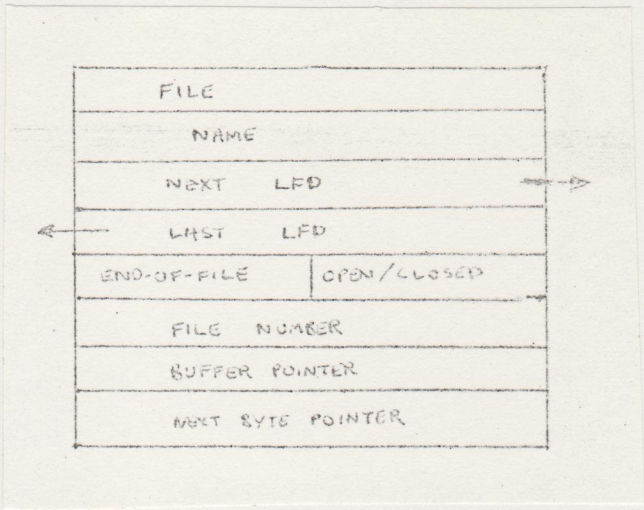

Fig 3.6 Local file descriptor

3.2.4.2 The i/o buffer

Because single bytes of data cannot be accessed on the disc a buffer has to be allocated to allow input and output to take place.

This must be 256 words long to enable the last word of a disc sector to be obtained.

The buffer is only used for the OPEN and READ/WRITE routines and is not necessary for the others.

3.2.4.3 The open file chain

When files are opened for reading or writing the lfds describing the files are attached to an 'open file' chain.

This is used for emergency exits from user programs and enables all open files to be closed in preparation for the next program.

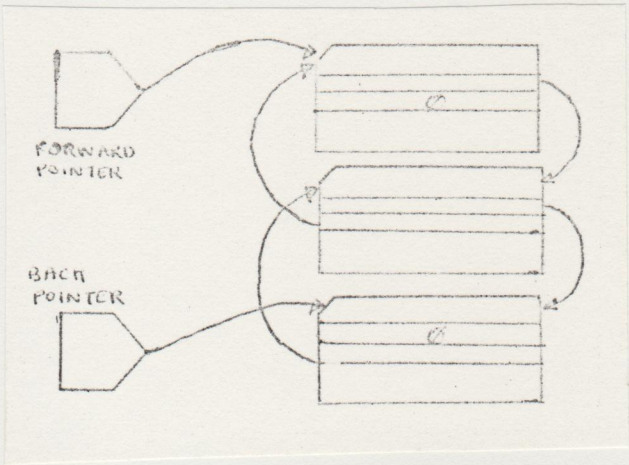

Fig 3.7 Open file chain during program run

3.3 Error handling

3.3.1 Non-fatal errors (warnings)

If a non-fatal error occurs a warning message will be displayed, on the command terminal, and the users program may continue unhindered.

Non-fatal error messages can be suppressed by setting a warning inhibit flag.

Examples of non-fatal errors are:-
"?? delete- no file"
"?? close- file closed"

3.3.2 Fatal errors

Fatal errors result in the termination of the user program. All open files are closed and an error message is displayed on the command terminal. No return can be made to the user program and therefore control is returned to the command decoder.

Fatal errors come in two categories:-
1) System errors
2) Machine errors

3.3.2.1 System errors

These occur when system executive calls are made illegally. An example of this would be attempting to open a non-existant file for input. These result in an error message being displayed, telling which executive call and stating the error, and termination of the program.

3.3.2.2 Machine errors

These can arise when illegal instructions are used or traps or interrupts occur without the trap vectors being initialized. As before an error message is displayed and the users program terminated.

3.4 Command decoding

As described in section 2-2.3.1 command decoding is kept very simple. The file name input routine is used to read the name of the next file to be run. All unused characters on the rest of the line are available to the user.

3.5 Other programs

Many other programs are needed to get SUSIE operable and allow SUSIE to be tested.

3.5.1 Initialization

3.5.1.1 Disc initialization

This program has three functions:-

1) To put a copy of SUSIE on the disc,
2) To set up a blank file directory on the disc.
3) To set up the bitmap on disc such that the blocks holding the bitmap, file directory and SUSIE cannot be overwritten by user programs.

3.5.1.2 File initialization

After disc initialization there are no files available for execution. The file initialization program (LOAD) is used to place programs onto SUSIE. This program varies from the usual programs that run under SUSIE in that it must reset stack pointers and set interrupt enable bits for

keyboard input. This is necessary because the manual loading of LOAD (using the absolute loader) causes an INIT signal on the PDP UNIBUS - see refs [4, 5]. 'LOAD' is best used to place itself onto the disc.

3.5.2 Bootstrap

The bootstrap program is used to get the SUSIE operating system off the disc and start it running.

3.5.3 Utility programs

One utility program has been written to allow files to be maintained on the disc. Its operation is described in section 4-5.

3.5.4 Test programs

Two test programs are used to test the two main areas of the operating system.

1) File test [FT] is used to test all file operations except INDEX (this is tested in the utility program). Its use is shown in the sample runs.

2) Keyboard test [KT] is used to test the routines that have not been used directly by SUSIE. These include ECHO, WARNING INHIBIT and DELAY.

4. Assembling

Normal assembly procedures are followed for assembling SUSIE. The system must be assembled as an absolute image (c.f. link loading).

Having run MACRO the following command string must be used:-

```
PP:/EN:ABS,<SUEDSP,SUEKBD,SUEDSC,SUECOM
```

Section 4

Using SUSIE

Nigel Eke

21-May-79

Section 4

Using SUSIE

1. Initialization of a disc

The following steps show how a disc has to be initialized. Any reference to 'NAME' indicates a paper tape titled 'NAME'.

1) Use the absolute loader to run 'DKINIT'.

2) Place tape of 'SUSIE' in tape reader.

3) Press CONTINUE.

4) Use the absolute loader to run 'BOOT'.

5) Once SUSIE has started HALT the machine.

6) Use the absolute loader to run 'LOAD'.

7) LOAD is now requesting a file name to be loaded on to disc. It is best used to load itself. (Use CNTRL -K to end entry of a line).

8) LOAD can now be used to load other programs such as the file test tape 'FT' or the utility program 'FIDO'.

2. Running programs on SUSIE

1) Use the absolute loader to run 'BOOT'. This will start SUSIE running - indicated by:-

```
HI THERE...
SUSIE AT YOUR SERVICE.

#
```

being output to the display.

2) When SUSIE places a '#' on the display this indicates the system is expecting the name of a program to be run.

Any line of data input to SUSIE must be terminated by <end of line> (CNTRL -K).

Delete (<DEL> or rubout) and cancel (<CAN> or CNTRL -X) can be freely used on input with their normal results.

3. Preparing programs for SUSIE

3.1 The DOS-BATCH system

The DOS-BATCH system is currently used to prepare object programs for SUSIE. The DOS-BATCH editor is used to prepare the SUSIE source program and the MACRO-11 assembler is used to produce an object tape.

3.2 SUSIE system macros

A file has been prepared on the DOS-BATCH system that MUST be concatenated to the start of the user program. Together with initialization lines (register descriptions etc) there are also included a number of macros to simplify the understanding of executive calls (See part 4).

3.3 User source programs

The source file of a program for SUSIE is very similar to normal MACRO-11 programs. It must be noted that NONE of the DOS-BATCH system macros or special assembly directives (such as .PSECT) must be used.

The general form of the source file is

```
$NAME <title of program>
      .
      .
.END start address
```

Register 5 must NOT be used as a general register. (See part 4).

3.4 Assembling the source program

An object tape can be punched by using the MACRO-11 assembler in the following manner:-

```
PP:/EN:ABS,LP:<SYSMAC.SUE,user file name.ext
```

See ref [6] for details of the above switches. Note the compulsory use of the file SYSMAC.SUE (SUSIE's system macros).

3.5 Placing programs onto SUSIE

'LOAD' can be used to put the user programs onto SUSIE's operating system.

1) Run SUSIE

2) Put object tape in reader
3) Run LOAD and enter name of program
4) Enter blank file name (to exit from LOAD)
5) Type name of program to start it

4. System macros

A number of system macros are available to the user to make the executive calls simpler and clearer to understand. Any information returned by these executive routines is returned on the parameter stack. The stack pointer for this is register 5 (PSP) and must only be used as such. It should only be used for 'pushing' and 'popping' words of data. (I.e. MOVB #'Z, -(PSP) is ILLEGAL).

A few macros are available to aid end-of-file testing and message creation. The use of all these macros is described in detail here:-

4.1 Name

Call by:-
```
$NAME <Title>
```

Produces title on list file.

4.2 Mess and Lmess

Call by:-
```
$LMESS <Message> or
$MESS <Message>
```

Produces string of the form:-
```
.WORD number of characters in message
.ASCII /message/
.EVEN
```

LMESS places a newline in front of the message. The character '/' must not be used in the message.

4.3 Lfd and Lfdb

Call by:-
```
$LFD <fname>, buffer address or
$LFDB <fname>
```

Produces a local file descriptor. LFD assumes the 256 word buffer is defined elsewhere. LFDB defines the buffer

space immediatly after the local file descriptor and should be used in preference to LFD.

The filename MUST BE six alphanumeric characters.

4.4 Exit

Call by:-
$EXIT

Makes executive call to exit from program.

4.5 Delay

Call by:-
$DELAY delay value

Delays program by 'delay value' 50ths of a second.

4.6 Cpos

Call by:-
$CPOS

Returns value of the current cursor position.

4.7 Ochar

Call by:-
$OCHAR character, new cursor position

Outputs character directly to command device and sets up new cursor position.

4.8 Dchar

Call by:-
$DCHAR character

Outputs character to command device.

4.9 Dtext

Call by:-

$DTEXT text address

Displays text on command device. Text is of the form produced by MESS and LMESS.

4.10 Swrni

Call by:-
$SWRNI flag

Sets warning inhibit flag.
=0 warning messages are output
#0 warning messages inhibited

4.11 Warn

Call by:-
$WARN text address

Produces warning message on command device. The text is of the form produced by MESS and LMESS.

4.12 Error

Call by:-
$ERROR text address

Produces error message on command device. The text is of the usual form (See MESS and LMESS). Results in the termination of the user program.

4.13 Tchar

Call by:-
$TCHAR

Tests to see if character available for input. Returns
0 if no character
char if character

4.14 Inchr

Call by:-
$INCHR

Waits for character to become available and returns it.

4.15 Tline

Call by:-
$TLINE

Tests to see if line available for input. Returns
0 if no line
next char if line

4.16 Inlin

Call by:-
$INLIN

Waits for line to be input. Returns next character in that line.

4.17 Ineol

Call by:-
$INEOL last character input

Tests last character input to see if it is an end of line. If not then it inputs characters until end of line is found.

4.18 Infnm

Call by:-
$INFNM lfd address

Inputs file name from command device and places it in the local file descriptor. If no legal characters in the file name then the lfd is not disturbed. (This can be used for default file names).

Returns first illegal character to be read.

4.19 Steco

Call by:-
$STECO flag

Sets up echo flag:-
=0 no echo when characters input
#0 characters are echoed

4.20 Sesci

Call by:-
$SESCI flag

Sets up escape inhibit flag:-
=0 <esc> forces program termination
#0 prevents <esc> terminating program

4.21 Look

Call by:-
$LOOK lfd address

Returns information about the file described in the local file descriptor. If the file does not exists then it returns 0 only. If file exists then it returns:-
non zero value
backup existance (=0 no backup)
open/closed (-ve - open for writing
0 closed
+ve - open for input)

4.22 Openi and Openo

Call by:-
$OPENI lfd address or
$OPENO lfd address

Attempts to open a file, described by the local file descriptor, for input/output. A successful attempt results in return to users program.

4.23 Read

Call by:-
$READ lfd address

Returns the next character from file.

4.24 Write

Call by:-
$WRITE lfd address, character

Writes the character to file.

4.25 Close

Call by:-
$CLOSE lfd address

Closes file described by the local file descriptor. The local file descriptor can now be used for other file control executive calls.

4.26 Breof and Brneof

Call by:-
BREOF lfd addr, branch address
BRNEOF lfd addr, branch address

Used to detect end of file on reading. BREOF will branch to 'branch address' if end file has been found on previous READ. BRNEOF will branch if end of file has not yet been found.

4.27 Delete

Call by:-
$DELET lfd address

Will delete the main copy of the file described by the local file descriptor. Any backup copy will then become the main copy.

4.28 Rename

Call by:-
$RENAM lfd address 1, lfd address 2

File described by lfd1, will be renamed. It will then have the same name as that of lfd2.

4.29 Index

Call by:-
$INDEX file index number (n)

Returns information on the n th file in the file directory. On return the data on the stack is:-
high order name (3 RAD50 characters (See ref [6]))
low order name (3 RAD50 characters)
main file existance

backup file existance
open state (=0 - closed)

All data is returned onto stack regardless of the state of the actual data (c.f. LOOK).

4.30 Run

Call by:-
$RUN lfd address

Runs the file described in the local file descriptor. NO return is made to the users program. This routine results in an error exit or a successful run of the next program.

5. File utility program (FIDO)

5.1 General description

FIDO (FIle Directory Operator) allows control of the creation, deletion and listing of files on SUSIE. It can be run by typing FIDO<end of line> in response to the system prompt.

5.2 Commands

Having entered FIDO it will expect one of a number of commands to be typed. These are described below.

5.2.1 File transfer

To obtain a new copy of a file type:-
new file<old file

Leaving NEW FILE field blank will produce a listing on the command device.

Leaving the OLD FILE field blank will allow input from the command device. (The character <SUB> (control Z) will terminate this input).

5.2.2 Renaming a file

To rename a file type:-
old file name<new file name/R

5.2.3 Deleting a file

To delete the main copy of a file type:-
file name/D

5.2.4 Running a file

To run another program while in FIDO type:-
file name/R

5.2.5 File indexing

To check the existance of a particular file type:-
file name/I

Leaving the FILE NAME field blank will produce an index of all the files on the system.

On the listing a '*' by the side of a file name indicates that a backup copy also exists.

5.3 Command concatenation

Multiple commands can be entered by separating each command by ','. For example:-
FRED/D,/I
will delete a file called FRED and follow this with an index.

Section 5

Project summary

Section 5

Project summary

1. Objectives

The object of this project was to write a small operating system aimed towards users on the amateur side of the personal computer market. Because of this the operating system must:-

1) be simple to use
2) have relativly small memory usage
3) have clear error messages
4) be fool proof

2. Achievements

All the objectives stated above have, to the authors knowledge, been satisfactorily completed.

2.1 Simplicity of use

This comes under two areas (both described elsewhere):-

1) Command structure (Sec 2-2.3.1 & 4-2.)
2) System macros (Sec 2-2.3.2 & 4-4.)

2.2 Memory size

The operating system currently uses 5 1/2 K bytes of memory. Because the whole executive is permanently resident in memory this is thought to be a reasonable size, although some further work could be made in this area (See 3.1).

2.3 Error messages and error handling

Attempts have been made to make the error messages as concise as possible without clouding their meaning. Error messages remain in English and have not reverted to the numbers so often found in operating systems.

For example:-

If an attempt is made to open a non existent file the message:-

```
!! OPENI - No file
```

will appear rather than

```
F001 132756 or
WHAT ???
```

Other errors that may occur in user programs are also dealt with. These include:-

1) Executing illegal instructions
2) Addressing odd word boundaries
3) Infinite program loops

One and two result in a hardware trap. The service routine for these will terminate the user's program.

Three is handled by allowing the <ESC> key (Control shift K) to be detected on input from the keyboard. When detected the user program will be terminated.

2.4 Dealing with the 'idiot' user

It is the author's belief that no system is totally fool proof and any serious attempt to 'break the system' will eventually succeed. Since this system is intended for a single user it is thought unlikely he will wish to destroy his own files etc...

Some precautions against accidental file corruption have been made- namely the use of automatic backup files.

Overwriting the local file descriptors (Sec 3-3.2.4.1) will probably result in file corruption but access to these is only intended for system programs. The user should access his file through the utility programs or interpreters and these should protect the file descriptors from being overwritten.

No programs can be loaded on top of the system residing in memory. This allows the file directory and other tables to remain intact.

3. Future work for the project

3.1 Memory size

Some work could be carried out concerning the optimization of the assembly code produced for this system. This should enable the operating system to fit into 4-5K bytes of memory.

With cheap memory available at the moment this is not an important task and would be completed purely for the authors satisfaction.

3.2 Contiguous files

The current file structure relies on a chain of file blocks to form a file. Because of this no random access files

can be created on this disc. It would be fairly simple to allow the user to allocate himself a number of contiguous blocks on the disc for his own use (E.g. random access files).

Although relativly simple to implement it is not thought necessary because the majority of amateurs will not require this facility, but this is a debatable point and worth further investigation.

3.3 Interpreters

To allow the user to write programs on the system (without reverting to DOS-BATCH) it would be very desirable to have a BASIC interpreter resident.

Although this work is not directly connected with the operating system it would increase the market of available 'ready to use' systems for the computer hobbyist.

3.4 Multiprogrammable systems

SUSIE (Single USer Interactive Executive) could be upgraded to a multiprogrammable or multiuser system (MARIE - Multiprogrammable And Real-time Interactive Executive).

Four areas need to be changed/added for an efficient upgrade to take place:-

1) Scheduling	Easily added by using the programmable clock to enable time-slices to be allotted to each users program.
2) Protection	File protection between individual users is necessary. Memory protection between running programs is also needed. This can be done by using memory management hardware.
3) Programming	Systems programs must be made re-entrant to cut memory usage. User programs must be made relocatable to aid replacement and memory allocation.
4) Input/output	For multiuser systems, control of the individual user's command terminals must exist providing all the facilities currently available for the command device. The disc control also needs to be largely rewritten since the system cannot

now wait for disc accesses to be completed before continuing.

It would be possible, although very inefficient, to write a multiprogrammable system to run under SUSIE. This would probably be a first step towards finally upgrading SUSIE.

Section 6

Appendices

Section 6

Appendices

1. References

[1] A dictionary of Computers - Penquin
[2] Computer organization & programming - Gear -Mcgraw-Hill
[3] OMNI - Nov 78; Computer lib - T Nelson pp58-62
[4] PDP-11/45 Processor handbook
[5] PDP-11 Peripherals and interfacing handbook
[6] D.E.C. DOS/Batch Handbook

2. Acknowledgments

Thanks to all at ASWE for their aid in the preperation of this report especially Dr W Lakin, Mr J Miles and Mr M Moulding.

3. Computer printout

The following pages comprise the project listings and sample runs of the system.

For the sample runs all user input is underlined. All unechoed input is handwritten and underlined. The <end of line> is indicated by .

Listings include:-

Susie	display version
Dkinit	disc initialization
Boot	bootstrap
Sysmac	system macros
Load	initial loader
Fido	file utility package
Kt	keyboard test program
Ft	file test program
Gbig	large file test

Sample runs include:-
- disc initialization procedure
- keyboard test
- file test
- large file test