JOHN TAYLOR

CONTENTS

System Programming Facilities

CHAPTER 1. INTRODUCTION TO THE SYSTEM

MU6 has a general-purpose operating system (known as MUSS) which has evolved from the design of the MU5 operating system. Its development has been characterised by a strong emphasis on producing an efficient and compact system. At the same time, ease of modification has been an important objective, and it is expected that individual users and installations will modify and extend the system to suit their own needs.

The central part of MUSS which would be common to all installations is known as the MUSS Basic System. It is designed primarily for use in an interactive computing environment although it also allows for the specification and input of complete jobs as in a batch system.

The rest of this chapter is a short description of the organisation of the MUSS system, in order to introduce the essential background information and terminology.

1.1 PROCESSES

From the user point of view the software in the machine can be regarded as consisting of a number of concurrent activities, or processes, for example

control of the lineprinter
management of the system accounts
execution of a user job.

In principle each of these processes can be thought of as executing within its own dedicated computer, but having some means of communicating with the other processes. However, it is a characteristic of many operating system activities, and user jobs, that they require the use of a processor for only relatively small amounts of time. The rest of the time is spent waiting for something to happen. For example, waiting for the lineprinter to finish printing the current line, for a user process to supply more output for printing, or for an on-line user to type his next line of input. Consequently, it is possible for all of the processes to share the use of a single processor. A part of the operating system known as the "kernel" allocates the processor to processes are required, giving each process the impression that it has a complete machine to itself.

between the process in a virtual machine and the outside world. Peripheral devices are controlled by special processes called device controllers, which communicate with other processes via the message system. Thus for example, information to be printed on a lineprinter must be sent as a message to "LPT", the device controller in charge of the lineprinters.

1.4 SUPERVISORS

User jobs are introduced into the system by processes called supervisors. A supervisor is a process which services requests (in the form of messages) from users to start jobs. Its main function is to create and start new processes to execute these user jobs, but it can also exert some control over the execution of any process which it has created by use of the appropriate operating system procedures.

There is a basic supervisor permanently resident in the system called JOB. In addition, any other process may act as a supervisor to provide alternatives to the basic system facilities. It was anticipated originally that this facility would be used to provide several alternative supervisors, each with its own job control language, specialised towards the needs of different user groups. In practice this has not been necessary because the basic system has proved to have sufficient flexibility for all users.

The role of the basic supervisor is simply to create a virtual machine to run a process. This process then interprets its own job control commands. In fact, the job control commands take the form of calls on library procedures which are to be made within the virtual machine running the process. Differing user needs are met by the variety of procedures available. Some procedures might in practice be interpreters of other job control languages. However, even this facility has not been exploited because job control procedures, like other library procedures, can be called from programs written in the high-level languages and complicated job control sequences involving conditional and repeated actions can be written in the standard high-level languages.

1.5 THE SYSTEM LIBRARY

Pre-loaded into every virtual machine created by the MUSS is a set of fully compiled library procedures. These provide the process with access to all of the facilities of the system, and include

mathematical functions
basic input/output procedures
compilers
editors
job control procedures

JCL interpreters
operating system interface procedures
('SEND.MESSAGE', for example).

The average user would not normally have contact with the operating system interface procedures. They are used by the supervisors, the basic input/output procedures which interface the users read and print commands into the message system, and the job control procedures. The job control procedures are mainly concerned with defining the environment in which programs are to run. This usually means defining the 'documents' which form the inputs and outputs of a program.

CHAPTER 2 USING THE SYSTEM

This chapter gives an introduction to the simple use of the MUSS operating system. More detailed descriptions of particular areas are given in later chapters. Because MUSS is primarily an interactive system, the emphasis in this chapter is on interactive operation, but all of the commands described can be used equally well in both batch and interactive contexts.

2.1 INTERACTIVE OPERATION

2.1.1 General

The MUSS Basic System is designed primarily for interactive use via serial teletype-like devices, and the system normally responds to input only after a complete line has been entered. A line of input is terminated by typing either a carriage return (CR) or a newline (NL) character; in the former case the machine will precede the next output by a newline. After processing a line of input, the system usually responds with a prompt requesting further input. In general this prompt may be any sequence of characters, but a number of standard prompts are defined for certain common situations. These are:

LOGIN: The terminal is currently idle, and a valid logging in command (see 2.1.2 below) must be given before it can be used.

** The terminal is logged in, and the system expects the next line to be a job control command (see 2.3 below).

-> This prompt is used quite commonly in cases other than the above.

A further common convention is that when text is being input, during editing or in the NEW command, the system prompts with the character which will terminate the text input.

At any time while a line is being input, the last character typed may be deleted by typing the backspace character (CONTROL H). This may be repeated as often as necessary to delete further characters, as far as the start of the line. The entire line may also be deleted by inputting the character CANCEL (CONTROL-X).

A user's job may be interrupted at any time by pressing the BREAK

CHAPTER 2 USING THE SYSTEM

This chapter gives an introduction to the simple use of the MUSS operating system. More detailed descriptions of particular areas are given in later chapters. Because MUSS is primarily an interactive system, the emphasis in this chapter is on interactive operation, but all of the commands described can be used equally well in both batch and interactive contexts.

2.1 INTERACTIVE OPERATION

2.1.1 General

The MUSS Basic System is designed primarily for interactive use via serial teletype-like devices, and the system normally responds to input only after a complete line has been entered. A line of input is terminated by typing either a carriage return (CR) or a newline (NL) character; in the former case the machine will precede the next output by a newline. After processing a line of input, the system usually responds with a prompt requesting further input. In general this prompt may be any sequence of characters, but a number of standard prompts are defined for certain common situations. These are:

LOGIN:	The terminal is currently idle, and a valid logging in command (see 2.1.2 below) must be given before it can be used.
**	The terminal is logged in, and the system expects the next line to be a job control command (see 2.3 below).
->	This prompt is used quite commonly in cases other than the above.

A further common convention is that when text is being input, during editing or in the NEW command, the system prompts with the character which will terminate the text input.

At any time while a line is being input, the last character typed may be deleted by typing the backspace character (CONTROL H). This may be repeated as often as necessary to delete further characters, as far as the start of the line. The entire line may also be deleted by inputting the character CANCEL (CONTROL-X).

A user's job may be interrupted at any time by pressing the BREAK key on the terminal. The effect of doing this depends upon what the job was doing at the time. In many cases the current activity will be abandoned, and the system will prompt for a new job control command. Another form of interrupt can be caused by typing the XOFF character (CONTROL S) while the job is outputting on the terminal. The effect of this is to halt the output, and either an EON character (CONTROL Q) to continue the output, or a BREAK.

2.1.2 Logging in

Before using the system, a user must identify himself by logging in. The normal logging in command is

```
***M JOB username password jobname options
```

Where 'JOB' identifies the supervisor to be used, 'username' and 'password' identify the user who is logging in, 'jobname' is the name the user wishes to call his job. All of this information must be supplied, but the user may also optionally supply a time limit and a priority for his job. For an interactive job, priorities in the range 8-11 should be specified, where 8 is the highest priority; if no priority is given, priority 11 will be used. High priority should result in a faster response relative to other jobs, but high priority jobs are charged for their processing time at a higher rate than low priority ones.

Provided that the logging in command was satisfactory, the system will respond by printing the jobname, time and date, and a prompt for a job control command (**).

examples:

```
 ***M JOB X03 XPMGRZS X03ED
 ***M JOB X03 XPMGRZS X03COMP T1000
 ***M JOB X03 XPMGRZS X03FAST P8
 ***M JOB X03 XPMGRZS X03SHORT P8 T10
 ***M JOB X03 XPMGRZS X03LONG T10000 P10
```

2.1.3 Logging out

To log out from the system, the STOP command is used. This prints the time and date, and the amount of processing time used, and sets the terminal into an idle state so that a subsequent line of input will result in the 'LOGIN:' prompt.

2.2 BATCH OPERATION

All of the basic system commands of MUSS can be used in batch mode as well as interactively. The output from a batch job is normally sent to a lineprinter close to the point where the job was submitted. The only changes required to run a job in batch mode are

i) The first line should be identical to the logging-in command described above, except that it begins with ***A instead of ***M.

ii) Priorities 12-15 should be used for batch jobs, with 15 meaning lowest priority. If no priority is specified, 14 will be used.

iii) The STOP command should be followed by '***Z' on a line by itself.

Example of a complete (but useless) batch job:

```
***A JOB X03 XPMGRZS X03COMP2

STOP

***Z
```

2.3 JOB CONTROL COMMANDS

After successfully logging in (or after the '***A' line of a batch job), the user types commands which direct the execution of the job. The interactive system prompts with '**' whenever it is expecting such a command.

All of the commands which may be used are procedures in the system library (or in some private user library, see Chapter 8). In fact, any procedure in the system library can be used as a job control command. Conversely anything which can be used as a command can also be called as a procedure, by programs in any of the programming languages supported by the system. For more details, refer to Chapter 8 of this manual.

2.3.1 Command Format

A command is input as the name of the command to be called, followed by its parameters if any.

```
COMMAND PARAM1 PARAM2 ... PARAMn
```

The command name and its parameters are separated by spaces and the entire command is terminated by a newline. Rightmost parameters may be omitted, in which case zero will be substituted; most of the commonly-used commands will give a suitable default value in this case.

Command Examples (the parameter forms are explained below).

```
    LISTFILE PROG LPT*  lists the file PROG on the lineprinter
    LISTFILE PROG       lists the file PROG on the terminal
    LISTFILE 0 LPT*     lists the "current file" on the lineprinter
    LISTFILE            lists the "current file" on the terminal.
```

In the command descriptions in this manual, many of the command names contain the character '.', for example LIST.FILE. This is for readability only, and is not actually required, and '.' typed anywhere in a command name will be ignored.

Optionally, the command name may be preceded by two asterisks

```
**LISTFILE PROG
```

This form is normally only used if commands are to be executed during compilation of a program; all of the system compilers recognise lines beginning with '**' and interpret such lines immediately as job control commands.

2.3.2 Command Abbreviations

All of the commands are in fact procedures in the system library. Some of these procedures have a unique two or three letter abbreviation which may be used instead of the full procedure name. For example, LISTFILE may be abbreviated to LF. The abbreviations are given with the command descriptions in the manual, and also in the summary of commands in Appendix 1.

2.3.3 Parameters

Most of the procedures in the system library have parameters and results of only three different types, denoted in this manual as I,II, and C. I represents a single length integer value equivalent to the type INTEGER in MUSL; II represents a 64-bit item (LOGICAL64 in MUSL) which in most cases is used to pass an 8-byte packed character string; C represents a single character (LOGICAL8). In addition to these three basic types, vectors may be specified (denoted [I], [II] and [C]), and also pointers or ADDR type variables (denoted ^I, ^II and ^C). The distinction between [I] and ^I is that [I] represents a specific number of integer elements, whereas ^I merely points at one such object and can be used to access any number of integers starting at that point. A fourth type of parameter, a pointer to a procedure, is also occasionally used. This is denoted by the letter P.

Within the command descriptions, the types of the parameters are indicated by the command heading, e.g.

```
LIST.FILE([C],[C],I,I)
```

means that LIST.FILE has four parameters, of which the first two are vectors of characters and the last two are integers. The parameters are referred to individually as P1, P2, etc. Some command descriptions may also refer to a set of global variables PW0, PW1, etc (of type I) and PWW0, PWW1, etc. (of type II). These are sometimes used for passing results back from library procedures and are not usually of importance in job control contexts, with the exception of PW0 which returns status information on completion of a command. If PW0 = 0, the command completed successfully; otherwise PW0 contains a fault number. The fault numbers are listed in Appendix 2, but the command interpreter will normally output a suitable error message if such a fault is detected.

Job control parameters may be written in three different ways: as a decimal integer, a hexadecimal value, or a character string.

The decimal integer form is the normal representation for parameters of type I. If it is used for either of the other two parameter types, it will be interpreted as a character string of decimal digits. An exception is the decimal integer '0', which is always treated as a numeric zero (this allows zero values to be supplied for parameters of type II and ^C, to obtain default actions). The decimal integer may be preceded by a '+' or '-'.

The hexadecimal form may be used for any type of parameter. The hexadecimal number is preceded by '%', and repetition of a digit may be indicated by a positive integer in brackets (e.g. %F0(3)F is the same as %F000F). The number is evaluated as an II quantity, and converted if necessary to the required size.

Character strings are permitted for parameters of type II and [C]. For II parameters, the characters are packed, right justified, and zero filled into a 64-bit word. If too many characters are supplied, the last eight are taken. For a [C]-type parameter a descriptor to the whole string is created.

2.4 USING FILES AND DOCUMENTS

A high proportion of all job control commands, and many of their parameters, are concerned with setting up inputs and outputs. Physical input/output exists as "documents", which may take a variety of forms: for example, a deck of cards; a lineprinter listing; a file, or a "conversation" at an interactive terminal. Within a job, documents are assigned to streams for processing, using commands which are described in detail in Chapter 3. However, many of the most commonly used system commands have parameters which relate directly to input and output documents, and assign these documents automatically to streams for the user. This kind of parameter is referred to as a "document name".

2.4.1 Document Names

Generally there are three main "kinds" of document which may be used, and three corresponding forms of document name. These are:

i) A file, in this or another user's filestore. This is normally written as a filename of up to 8 characters, the last of which must not be '*'. If another user's file is to be accessed, the file name is followed by '/username with no intervening spaces. In this case, of course, the file owner must have given permission for this user to access the file (see Chapter 5).
Examples: EDIT FILE1 FILE2
FORTRAN FORTPROG/GRF

ii) An input/output device, or process. Output can be directed to any process in the system, by giving the process name followed by '*' as the document name. This is mainly used to send output to device controllers, such as LPT* (the lineprinter), PTP* (paper tape punch). In a multicomputer system, if the process is in another machine its machine identifier will normally also have to be appended.
Examples: LISTFILE FILE1 LPT*
LISTFILE FILE1 LPT*/CSD
If output is to be discarded completely, the document name * may be used.

Normally only supervisors accept input directly from devices and other processes. For this, the DEFINE.INPUT command described in Chapter 3 must be used.

iii) A stream which has already been assigned a document, using the DEFINE.INPUT or DEFINE.OUTPUT commands described in chapter 3. In this case the special name STRn* (for stream n) may be used.
Example: LISTFILE FILE1 STR2*

2.4.2 The Current File

There is one further kind of document, called the current file. This is like a file, but is exists only for the duration of the job unless some explicit action is taken to save it as a permanent file. The corresponding document name is zero, and so it can be used as a default parameter that is, if no document name is specified by a command, then the current file will normally be used instead.

At the start of a job the current file is undefined. Certain commands, described later in this chapter, allow the user to set and alter the current file. Once it has been defined, omission of an input document name parameter will automatically result in the current file being used instead. If an input document name is omitted when no current file is defined, or when it has already been assigned to another stream, the currently selected input stream is used by default instead.

In the case of output documents, omission of the parameter causes the current file to be updated with whatever is output. There are a few commands which do not follow these general rules. For example, LISTFILE always defaults its output to the currently selected stream if the parameter is omitted.

2.5 COMMONLY-USED COMMANDS

2.5.1 File and Current File Commands

These commands allow the user to manipulate files and set up the current file. To access another user's files, the form "filename/username" should be used (see Chapter 5).

1) NEW([C],C)

This command is normally used to input text files. P1 is the name of an output document to be created, and P2 is a single character terminator. Input is copied from the current stream to the specified document, terminating when a line is found which starts with the terminator. If the terminator is omitted, '/' is used. The system prompts for more input using the terminator as prompt.

2) OLD([C])

This command designates a file, specified by P1, as the current file.

3) SAVE([C])

This command preserves the current file as a permanent file. P1 gives the file name. If a file of this name exists already, it will be replaced. The file continues to be the current file.

4) DELETE([C]) DEL

This command is used to erase a file (P1). If P1 is zero, the current file is deleted.

5) RENAME([C],II) REN

This command is used to change the name of an existing file. P1 specifies the old file name, which may be in another user's directory, P2 specifies the new name. If the new file name already exists in the directory, an error is signalled.

6) LIST.DIR(C,[C]) LD

This command lists, on the currently selected output stream, the files in the current user's directory.

P1 is used to indicate the level of detail required in the directory listing, as follows:

P1 = 0	File limits and list of file names.
P1 = "N"	File limits and list of file names.
P1 = "L"	File limits only.
P1 = "A"	All information.

P2 may be used to select a subset of the files in the directory. If P2 is non-zero, only files whose names contain the string P2 will be listed.

7) LIST.FILE([C],[C],I,I) LF

This command is used to list a text file with page and line numbers. P1 specifies the document to be listed, and P2 the destination. If P2 is zero, the listing is on the currently selected output stream.

If the last two parameters are zero, the whole file is listed, but they can be used to specify a first and last line.

8) COPY.FILE([C],[C],I,I) CF

This command is similar to LIST.FILE, but page and line numbers are not output, and non-character files may be copied in which case P3 and P4 specify positions as returned by I.POS (see Chapter 4).

9) EDIT([C],[C]) ED

This command invokes the editor to modify a text file. A complete specification appears in Chapter 5.

10) LIST.PERMIT([C],[C])

This command lists, on the currently selected output stream, the file permissions associated with the current user's directory.

P1 may be used to select a subset of the permissions. If P1 is non-zero, only files whose names include the string P1 will be listed. Similarly, P2 may be used to select a subset of the user names for whom permissions have been granted.

2.5.2 COMPILING AND RUNNING PROGRAMS

Additional instructions for compiling programs are given in Chapter 8 onwards. Here the rules for simple jobs are summarised.

1) LANGUAGE([C],[C],I,I)

The commands for the standard system compilers FORTRAN, COBOL, PASCAL, MUPL, MUSL, are all of this form with the appropriate name substituted for LANGUAGE. P1 is the document name for the program to be compiled. P2 is the document name for the compiler output. The remaining parameters specify mode and library information, and may normally be left unspecified.

P3 has a bit significant encoding.

```
 ________________________________________
 |                                 |1|0|1|
                                    |   |
 COMPILE A LIBRARY------------------    |
                                        |
 HIGH LEVEL RUNTIME DIAGNOSTICS---------
 NOT REQUIRED
```

When compiling a library P4 specifies the maximum number of procedures permitted in the interface of the library. If P4 is zero a suitable system default is used.

2) RUN([C])

This command enters a program compiled by an earlier compilation command, provided that the compilation was error free. P1 is the document name to which compiler output has previously been sent.

3) LIBRARY ([C],I) LIB

This command loads a library P1 and links it in to the library directory structure so that its procedures may be used as library procedures. P1 is a public library name of a library other than the basic library or a document containing a private library. P2 is the mode in which the library is opened. The most recently loaded library directory is inspected first, therefore any user library procedures with the same names as system library procedures will take precedence over the corresponding system procedures. The facilities provided for creating library files are described in the later Chapters (9-11) that relate to specific compilers and in the MUTL Manual.

If P2 = 0 it indicates that the library is to be used in the 'normal' mode for the machine in question. Normal mode on machines with large virtual memories means that the library document is opened into the virtual memory. For machines with small virtual memories the norm is to overlay (MAP) the library document, P2 = 1 indicates that the library is to be permanently loaded.

4) DELETE.LIB ([C]) DL

This command unloads a user library (P1) and unlinks it from the system directory.

If P1 = 0 all libraries except the basic system library are unloaded.

2.5.3 Commands for job sequencing etc

1) STOP (I) STP

This command terminates the job, after disposing of its output streams. The current file is discarded. P1 is a "reason code" for the termination. Zero indicates normal termination. A negative value indicates an abnormal (error) termination, in which case any remaining file outputs are not updated (see Chapter 3. DEFINE.OUTPUT).

2) IN ([C])

This command causes job control commands to be read from the specified input document (P1). If P1 contains the string "-1" the previously selected input document is restored.

3) RUN.JOB ([C],[C],[C]) RJ

This command is used within one job to initiate execution of another as a separate (background) job. P1 is the name of an input document on which job control commands for the new job may be found. If it is left unspecified and no current file is defined, the commands for the new job are input from the current stream terminated by '/' at the start of a line. P2 is the name of the supervisor to which the job is to be submitted, in the form "process name*" or "process name*/machine name". If P2 is left unspecified, JOB is assumed. P3 is the "header" (i.e. ***A line) for the new job, with the "***A supervisor name" omitted. If P3 is left unspecified, a header line of the form "username password title" will be generated, where username and password specify the current user and title is a unique jobname.

4) KILL (II)

This command may be used by any user to kill one of his own jobs, or by the operator to kill any job. P1 gives the name of the job to be terminated.

5) PPC.SEQ (I)

This procedure processes one or more job control commands from the current input stream. If P1=0, commands are processed as for a batch job; with P1 nonzero, commands are processed as for an interactive job.

6) PPC.CMD

This procedure reads and processes a single job control command from the current input stream.

2.6 JOB CONTROL EXAMPLES

(1) A 'Null' Job

This is an example of a background job which does nothing useful, but it illustrates the small amount of red tape required by all jobs. The meaningful commands would be placed before the STOP command.

```
***A JOB  USER PASS NULLJOB
STOP
***Z
```

(2) A 'Null' Fortran Job

This job illustrates the structure required to compile and run an Fortran program. The actual program would be placed after the FORTRAN command and before the END statements. The *END statement is needed at the end of all programs submitted to the MUSS compilers in order to end the compilation and switch back to command mode. A temporary return to command mode, for example to select a new input stream, can be made by embedding commands preceded by '**' in the program text. If a program requires input data it should be placed between the RUN and STOP commands. A user program may return to command level by executing the final end.

```
***A  JOB USER PASS NFTN
FORTRAN
.
.
.
      END
*END
RUN
STOP
***Z
```

(3) A Fortran Job Using a File

This job illustrates two actions which would normally be used only by on-line users. The first is the creation of a file (FILEX) which is followed by a call on the Fortran compiler to compile the file, after which is a RUN command to run the program.

```
***A JOB USER PASS  FJOB
NEW FILEX
.
.
.
      END
*END;
/
FORTRAN FILEX
RUN
STOP
***Z
```

(4) A Fortran Job Using the Current File

The facility illustrated here would again be used by on-line rather than background jobs, but it suffices to illustrate the mechanism. It is similar to the previous example, except that the file name has been omitted in the case of both the NEW and FORTRAN commands, hence the current file is used. This ceases to exist when a job ends, unless it is saved as a permanent file by the SAVE command also illustrated here. It should be noted that if any command fails, those following will not be executed. Thus if the program is faulty the file will not be saved.

```
***A JOB USER PASS CFJOB
NEW
.
.
.
      END
*END;
/
FORTRAN
RUN
SAVE FILEX
STOP
***Z
```

(5) Saving a Compiled Fortran Program as a File

A compiled program can be saved, for subsequent running in a file specified as the second compiler parameter.

```
***A JOB USER PASS COMP
FORTRAN FILEX FILEY
STOP
***Z
```

In this example a program in a file FILEX is compiled and the binary code is saved in a file FILEY. The program can subsequently be run by giving FILEY as the parameter of the RUN command. For example

```
***A JOB USER PASS RUN1
RUN FILEY
STOP
***Z
```

If the program needs data it could appear after the RUN command. If it needs input/output streams other than the default (zero) they would be defined before the RUN command. A similar mechanism allows a private library of procedures to be compiled and filed. They can subsequently be used as commands or by programs and in effect are an extension of the system library.

(6) An Example Interactive Session

In the example given below the computer output is underlined to distinguish it from the user's input. On the actual system the distinction would be made by colour on devices which provide that facility.

The first command used after the log-in line is NEW, which is used to input to the current file a Fortran program, for computing prime numbers. This is followed by the FORTRAN command which compiles the program but finds one error. These are corrected by editing the current file. The first edit statement copies to line 16 and 'windows' the line. The second means

```
delete 1R
inset 'RI'
and window
```

Positions may also be selected by context but it is more convenient to use line numbers when a compiler gives them with the error reports. At the second attempt the program compiles correctly and it is entered by the RUN command. Since the program contains a call for the READ procedure which reads an integer, it prompts for data. When it is given the integer 20 it computes all prime numbers less than 10, and returns to command mode as a result of executing the STOP. However, a mistake in the program causes all ones to be printed. On examining the program it is clear that PRIMES(P) should have been set to I not 1 so this is corrected by editing. The program is recompiled and then runs correctly. At this point the current file is saved and listed, and the user logs out.

```
 LOGIN:***M JOB D4 XXX DEMON
DEMON 13.20.01  11.01.80
**NEW
/      INTEGER SIEVE(1000),PRIMES(200),LIM,P,J
/      WRITE(8,200)
/200   FORMAT(1X,'TYPE PRIMES LIMIT,I3 FORMAT PLEASE')
/      READ(8,100)LIM
/100   FORMAT (I3)
/      DO 1 I =1,LIM
/  1   SIEVE(I) = 0
/      P = 0
/      DO 2  I = 2,LIM
/        IF (SIEVE(I) .NE. 0) GO TO 2
/        P = P + 1
/        PRIMES(P) = 1
/        DO 3 J = I,LIM ,I
/  3     SIEVE(J) = 1
/  2   CONTINUE
/      WIRTE(5,201)LIM
/201   FORMAT(IX,'PRIMES UP TO ',I3)
/      WRITE(8,202) (PRIMES(I), I = 1,P)
/202   FORMAT(/(1X,8(I3,4X)))
/      STOP
/      END
/      *END
//
**FTN
** 1.16  STATEMENT NOT RECOGNISED
  1 ERRORS
    END 0000009D
       21 STATS COMPILED
**ED
->C16W
1.16)~^~      WIRTE(8,201)LIM
->D/IR/I/RI/
->W
1.16     WRI~^~TE(8,201)LIM
->E
 <CFILE> 13.33.58.  11.01.80. OK
**FTN
    END 000000B0

       22 STATS COMPILED
**RUN

TYPE PRIMES LIMIT,I3 FORMAT PLEASE
->020

PRIMES UP TO 20

  1    1    1    1    1    1    1    1


STOPPED:LINE1. 20
**ED
```

```
->C/PRIMES(P)/W
  .112)~^~      PRIMES(P) = 1
->D/1/I/I/W
1.12)    PRIMES(P) = I~^~
->E
 <CFILE> 13.38.07.  11.01.80. OK
**FTN
    END 000000B0

       22 STATS COMPILED
**RUN
TYPE PRIMES LIMIT,I3 FORMAT PLEASE
->20
PRIMES UP TO 20

  2    3    5    7    11    13    17    19




STOPPED:LINE1.20
**SAVE DEMO
**LF DEMO &
DEMO    13.39.26.  11.01.80.

1. 1      INTEGER SIEVE(1000),PRIMES(200),LIM,P,J
1. 2       WRITE(8,200)
1. 3   200 FORMAT(1X,'TYPE PRIMES LIMIT,I3 FORMAT PLEASE')
1. 4       READ(8,100)LIM
1. 5   100 FORMAT (I3)
1. 6       DO 1 I =1,LIM
1. 7     1 SIEVE(I) = 0
1. 8       P = 0
1. 9       DO 2  I = 2,LIM
1.10         IF (SIEVE(I) .NE. 0) GO TO 2
1.11         P = P + 1
1.12         PRIMES(P) = I
1.13         DO 3 J = I,LIM,I
1.14     3   SIEVE(J) = 1
1.15     2 CONTINUE
1.16       WRITE(8,201)LIM
1.17   201 FORMAT(1X,'PRIMES UP TO ',I3)
1.18       WRITE(8,202) (PRIMES(I), I = 1,P)
1.19   202 FORMAT(/(1X,8(I3,4X)))
1.20       STOP
1.21       END
1.22       *END
1.23  **STP
STOP REASON   0  COST   19  TIME  13.40.47.  11.01.80.
```

CHAPTER 3. INPUT/OUTPUT STREAM MANAGEMENT

3.1 INTRODUCTION

The input/output system was briefly introduced in Chapter 2. It was explained that physical input/output exists as "documents", which may take a variety of forms: for example, a deck of cards; a lineprinter listing; a file, or a "conversation" at an interactive terminal. The input/output facilities described in this Chapter and Chapter 4 provide programs with a common interface to all kinds of input/output document, so that the programmer need not be aware of the actual source or destination of a particular document, or even of its type. They also provide a common interface between the input/output facilities of all of the programming languages, so that a file generated by a FORTRAN program may be read by a COBOL program, and vice versa.

High-level language programs perform input and output operations using the facilities of the language in which they are written. These are described in the relevant programming language manuals. However, all of the programming language facilities in MU6 are implemented in terms of the basic operations described in the next Chapter and some languages use the basic operations directly. The exact relation of the programming language facilities to the basic input/output system is described separately for each language in the appropriate Chapter of this Manual.

Input and output operations in MU6 are organised in terms of logical input/output streams. Each process may have up to 8 input streams and 8 output streams in use at any one time, and may switch between these at will. The most important functions of the basic input/output system are to provide the means for

i) Assigning actual documents to input and output streams (usually by job control commands).
ii) Selecting the particular stream to be used for ea input/output operation.
iii) Performing actual input/output operations on a stream.

The basic operations provided are used by particular compilers to build up the complete set of input/output facilities for the corresponding programming languages. This Chapter describes the stream assignment operations; the others are described in Chapter 4.

3.2 CHARACTERISTICS OF INPUT AND OUTPUT STREAMS

3.2.1 TYPES OF INPUT/OUTPUT STREAM

The basic system supports four different methods for structuring input/output information, namely:

(i) As a sequence of characters, structured into pages and lines.

(ii) As an unstructured sequence of binary information.

(iii) As a sequence of equal-length records, called units. Internally, each unit may be regarded as a sequence of characters or binary information.

(iv) As a sequence of variable-length records, called records. Internally, each record may be regarded as a sequence of characters or binary information.

Additionally, there are two distinct ways in which a stream may be accessed, irrespective of how the information within it is structured. These are termed random and sequential; the precise meaning of the terms in this context will be explained later. The combination of data structuring techniques and stream organisations gives eight different kinds of stream. Corresponding to these are eight possible classes of input/output operations. In practice, however, random I/O operations are performed by first selecting the required position and then performing sequential I/O.

3.2.2 TYPES OF INPUT/OUTPUT OPERATION

FILE ORGANISATION	SEQUENTIAL I/O OPERATIONS
CHARACTER	IN.CH OUT.CH
BINARY	IN.BIN.B OUT.BIN.B IN.BIN.S OUT.BIN.S
UNIT	IN.UNIT OUT.UNIT
RECORD	IN.REC OUT.REC

Table 3.2-1 Summary of major I/O operations.

Table 3.2-1 summarises the major input/output operations available, and classifies them according to stream type. Obviously, the intention is that a given class of operations be used on a stream of the appropriate type. Thus for example, one would expect IN.BIN.S to be used on an input stream, with binary data.

To achieve an efficient implementation, the system does not check for compatibility between operations and stream type on every operation. Instead the user specifies, at the time of assigning a document to the stream, which stream type is expected. Compatibility is checked at this point, and the user is then expected to use only compatible operations. Furthermore, sequential character and binary operations may be used within individual units or records of unit or record structured streams, so that the compatibility rules are not quite as strict as the above would suggest.

3.2.3 SECTIONED STREAMS

The actual document assigned to a stream may be either a file (or sequence of files) or a message (or sequence of messages). Each file or message, if there are more than one, is referred to as a section of the stream. The division into sections is a physical convenience, and is not apparent to the program. However, random access operation apply only within the current section of a stream.

When a sequence of files is specified as the document to be assigned, the files in the sequence should have names obtained by incrementing the first file name. The incrementing is performed on the assumption that the last character in the name is a decimal digit. Thus, for example, if the first file is called "FILE01", an orderly sequence from "FILE01"" to "FILE99" may be used.

Production of sections for an output stream is controlled by two parameters of the stream, the section size and maximum number of sections, which are specified when the stream is first defined. A stream may also be broken into sections explicitly, using the BREAK.OUTPUT operation.

3.2.4 MESSAGE STREAMS - MESSAGES AND SYNCHRONISATION

A message stream consists of a sequence of messages from or to another process. Streams which communicate with peripheral devices are of this type, since the devices are controlled by special processes. Thus for example input and output on an interactive terminal uses message streams.

The system distinguishes between two types of message, called long and short messages. Long messages are used for communicating with bulk input/output devices (such as card readers and lineprinters), and short messages for interactive devices. The user need not be aware of the distinction in the case of input streams, as the system will automatically read from whichever kind of message is received. For output, however, the user must define which kind of message is to be used on any message streams he creates. If short messages are specified, the section size is determined by the size of the buffer used - about 100 characters - and no section limit is imposed.

In the case of message input streams, the user also has to define what action the system should take when an attempt is made to read beyond the last available message. There are two options: to wait until further messages arrive, or to signal a fault. The former is suitable for input from an interactive terminal, and such a stream is referred to as an online input stream. The latter is more appropriate for batch input, which is normally all present before the job is started; such a stream is called on offline input stream.

For message output streams, the user may also wish to indicate that the output is to be synchronised to the operations of the receiving process. In interactive operation, for example, it is undesirable for a large amount of output to be generated by a job in advance of its being printed. Two means of synchronisation are possible, one automatic and the other explicit. In the automatic case, the job is suspended when the message is sent, and can only be freed by the receiving process. This is used to synchronise with an output device, as the output device controllers will perform the freeing operation, if necessary, on completing the output of a document. The alternative method is to request the receiving process to send a message in reply; this message must then be detected and read explicitly. This option enables a job to request notification when output is performed, without actually being halted.

3.2.5 IO STREAMS

An IO stream is a stream on which both input and output operations may be performed. Usually, such a stream will be randomly accessed, but this is not necessary. For sequential operations, a <u>single pointer</u> is used for both input and output. Thus, interleaved sequential input and output operations will operate on consecutive elements of the stream.

An IO stream occupies both an input and an output stream. Releasing or re-defining either the input or the output associated with an IO stream causes both to be discarded.

3.3 ASSIGNING DOCUMENTS TO STREAMS

The eight input and eight output streams of a process are identified to the basic I/O system by stream numbers in the range 0-7. Input and output stream 0 are normally used for the "default" input and output stream associated with a job. For example a job submitted on cards will automatically have input stream 0 assigned to the card input document, and output 0 to the lineprinter associated with the input station; for an interactive job, input and output streams 0 are automatically assigned to the terminal input and output respectively. The remaining streams are assigned by the user to any files or other documents which are needed during the course of the job.

Documents may be assigned to streams either explicitly by the user, or automatically by the software he is using. In the case of standard system software such as compilers and editors, and of most applications packages, the assignment is automatic - the user supplies a "document name" as a parameter, and the corresponding document is assigned behind the scenes to a stream. The main use for explicit assignments arises in running a user's own programs. Here, the language implementer will specify the correspondence between the stream numbers of the operating system and the "logical channels", "files", or whatever are operated on by the program. It is then up to the user to assign documents to the appropriate streams prior to entering his program. Sometimes it is desirable to operate on an explicitly assigned stream using system software, and special document names are available to allow this to be done.

3.3.1 PROCEDURES FOR DEFINING AND RELEASING STREAMS

1) INIT.IO()

This procedure completely re-initialises all input and output streams, abandoning any which are currently defined. It is performed automatically at the start of a job and is not normally used again, but it can be used a a drastic recovery measure in the case of very serious errors.

2) DEFINE.INPUT(I,[C],I,I)I DI

This procedure defines a new input stream, overriding any existing definition of this input stream. If the stream specified is defined already, any input being processed (i.e. the current section) on the stream is discarded. If the currently selected input stream is re-defined, it automatically remains selected but with the new document assigned to it.

P1 specifies the stream number to be defined (0-7). If P1 is negative a free stream number will be selected by the system. The stream number actually used is always returned as the integer result of the procedure.

P2 specifies the document which is to be assigned to the stream, and may take any of the forms

(a) A filename, indicating that the document is a file, or a sequence of files if the mode (P3) indicates that continuation is permitted. If the filename contains the character '/', it will be interpreted as filename/username, and the file belonging to the user named will be accessed provided that permission has been given. If no username is specified, the currently selected directory is used.

(b) Zero, indicating that the current file is to be used. The current file can only have one section, and may only be assigned to one input stream at a time. If no current file is available, the currently selected input stream will be assigned instead provided that the mode (P3) allows this and that P1 was negative.

(c) The character '*', indicating that the document will consist of messages addressed to this process' message channel P1.

(d) A stream name - STR0*, STR1*, etc. This enables an already existing stream (already set up by a preceding DEFINE.INPUT) to be specified as a parameter. If P1 is negative, the stream number of the named stream is returned; otherwise the document on the stream named is re-assigned to stream P1 and the named stream becomes undefined.

P3 gives the mode to be associated with the stream, and is interpreted as follows

```
 _________________________________________________________
|                                  |i|h|g|f|e|d|c|b| a |
                                    | | | | | | | | |   |
(i) SEPARATE  SECTIONS         ---| | | | | | | |   |
(h) READ FROM HEADERS ONLY     -----| | | | | | |   |
(g) ONLINE STREAM              -------| | | | | |   |
(f) WAIT UNTIL FILE AVAILABLE ---------| | | | |   |
(e) EXCLUSIVE ACCESS REQUIRED -----------| | | |   |
(d) DEFAULT TO CURRENT FILE    -------------| | |   |
(c) FORCE COMPATIBILITY        ---------------| |   |
(b) UNKNOWN TYPE               -----------------|   |
(a) TYPE - 00 CHARACTER        ------------------   |
           01 BINARY
           10 UNIT (SIZE IN P4)
           11 RECORD
```

Notes

(a) (2 bits) Defines the type of input stream expected - see section 3.2.1

(b) Indicates that the type of stream is unknown and should be set from the type of the first section.

(c) Normally each section is checked for compatibility with the specified type, and the unit size if the type indicates unit structuring. Setting this bit causes the type check to be omitted. Not recommended!

(d) Indicates the action to be taken when the file name (P2) is zero and the current file does not exist or is unavailable. Zero means default to the current stream; 1 means signal an error.

(e) Indicates that exclusive access to the file is required. This means that while the stream is defined, no other process may use the file.

(f) Indicates the action to be taken when the file is unavailable because either another process has exclusive access to it, or exclusive access is requested and the file is already open. 0 means signal an error; 1 means wait until the file is available. In the latter case a fault may still be signalled if halting the process would lead to a deadlock.

(g) Indicates whether a message input stream is offline (0) or online (1); for a file stream, indicates whether continuations are allowed (1) or not.

(h) Indicates input should always be from the header, even for a long message - see section 3.2.4

(i) Indicates that sections of the stream are not to be merged - i.e. the change to a new section only takes place on an explicit call to BREAK.INPUT.

P4 gives the unit size, in bytes, if the mode indicates that the stream is unit structured.

3) DEFINE.OUTPUT(I,[C],I,I,I,I)I DO

This procedure defines a new output stream, overriding any existing definition of this output stream. If the stream is defined already, any output produced and not yet dispatched (i.e., the current section) is discarded. If the currently-selected output stream is re-defined, it automatically remains selected but with the new document assigned to it.

P1 specifies the stream number to be defined. If P1 is negative a free stream number will be selected by the system. The stream number actually used is always returned as the integer result of the procedure.

P2 specifies the document which is to be assigned to the stream, and may take any of the forms

(a) A filename, indicating that the document is to become a file, or a sequence of files if the section limit (P5) is greater than 1. If the filename already exists, the existing copy will be overwritten when the stream is broken. If the filename contains the character '/', it will be interpreted as filename/username and the file belonging to the user named will be updated provided that permission has been given. If no username is specified, the currently selected directory is used.

(b) Zero, indicating that the document, on completion, is to become the current file or output to the currently selected stream, depending upon the mode setting. The current file can only have one section. The "current stream" option is only valid for negative P1.

(c) A process name followed by a '*', indicating that sections of the document are to be sent as messages to the specified process. This option is used for output to standard system devices - for example LPT* for a lineprinter, PTP* for a paper tape punch. If the process name contains the character '/', it will be interpreted as process name/machine name, and the output will be sent to the machine named. Otherwise, the current machine is assumed.

(d) A stream name - STR0*, STR1*, etc. This enables an existing stream (already set up in an earlier DEFINE.OUTPUT) to be specified as a parameter. If P1 is negative, the stream number of the named stream is returned; otherwise the document on the stream named is re-assigned to stream P1 and the named stream becomes undefined.

(e) A reply name - REP0*, REP1*, etc. This indicates that any output produced is to be sent as a reply to the last section received on the specified input stream.

(f) The character '*' alone, meaning that any output produced on the stream is to be discarded.

P3 gives the mode to be associated with this stream, and is interpreted as follows.

```
 ______________________________________________________________
|                          l     | k   |j|i|h| | |d| | | a |
                           |       |     | | |     |       |
(l) DEST CHANNEL -----|       |     | | |     |       |
(k) SYNCHRONISATION ---------|     | | |     |       |
    %00 Unsynchronised               | | |     |       |
    %10 SUSPEND                      | | |     |       |
    %08 REPLY TO CHANNEL 0           | | |     |       |
    %09 REPLY TO CHANNEL 1           | | |     |       |
    etc                              | | |     |       |
(j) DISCARD ON ERRORS ------------| | |     |       |
(i) SAVE ON ERRORS -----------------| |     |       |
(h) SHORT MESSAGES -------------------|     |       |
                                            |       |
(d) DEFAULT TO CURRENT STREAM ---------------|       |
                                                    |
(a) TYPE - 00 CHARACTER ----------------------------|
           01 BINARY
           10 UNIT (SIZE IN P6)
           11 RECORD (MAX SIZE IN P6)
```

Notes

(a) Defines the type of output stream required - see section 3.2.1

(d) Indicates the action to be taken when the filename (P2) is zero. 0 means default to the current file; 1 to the current stream.

(h) Indicates output is to be produced in short messages (if a message channel) - see section 3.2.4

(i) See END.OUTPUT

(j) See END.OUTPUT

(k) (5 bits) Defines the synchronisation method, if any, to be used with the stream - see section 3.2.4

(l) (3 bits) Allows messages to be sent to a channel of the destination process other than channel zero.

P4 and P5 give, respectively, the maximum section size in K bytes and the maximum number of sections. For short message streams, these parameters are not used. A zero value for either gives an installation-defined default.

P6 gives the unit size or the maximum record size, in bytes, for streams which are defined by the mode to be unit or record structured.

4) DEFINE.IO(I,[C],[C],I,I,I,I)I   DIO

This procedure is used to define an IO stream, and assign input and output documents to it as required. If the specified stream exists already (either as an input stream, or an output stream, or both) the existing definition is overridden and any existing input/output is discarded.

P1 specifies the stream number to be defined. If P1 is negative, a free stream (i.e. a stream number for which neither input nor output is defined) will be selected by the system. The actual stream number used is always returned as the integer result of the procedure.

P2 specifies the input source, if any, for the stream. This may take any of the forms allowed for DEFINE.INPUT. If the stream name form, STR0*, STR1* etc, is used, the stream must already be an IO stream and P3 must specify the same stream name. An additional form, SCR*, specifies that there is no input document and a stream is to be created from scratch. The default to the current stream is not allowed.

P3 specifies the output destination. This may take any of the forms allowed for DEFINE.OUTPUT. If the stream name form STR0*, STR1* etc., is used, the stream must already be an IO stream and P2 must specify the same stream name. The default to the current stream is not allowed.

P4 specifies the mode for the stream, and is interpreted as follows

```
 ____________________________________________________
|          | l |  k  |j|i| |g|f|e| |c|b| a |
```

for fields (m)-(i), see DEFINE.OUTPUT
for fields (g)-(e), (c)-(a) see DEFINE.INPUT

P5 and P6 specify the section size and number of sections to be associated with the output part of the stream. P7 specifies the unit size or maximum record size, in the case where P4 indicates a unit or record organisation.

5) CHANGE.DEST(I,[I])

This procedure enables the destination process for an existing message output stream to be changed, without otherwise redefining the stream in any way. P1 gives the stream number whose destination is to be changed. P2 is a vector of four single-length (I) elements, containing the system process number (SPN), process identifier (PID) and channel number for the new destination, and a sequence number. This information may be found by calling IN.SOURCE - if the output is to reply to some input already received - or LOOK.UP.PROCESS - if the output is to an explicitly named process.

6) DEFINE.STRING.INPUT(I,[C],I,I)I DSI

This procedure enables a vector of bytes specified by P2 to be assigned to input stream P1, so that subsequent input operations on this stream read from the byte vector. If P1 is negative a free stream number will be selected by the system. The stream number actually used is always returned as the integer result of the procedure. P3 gives the mode, as for DEFINE.INPUT, and P4 gives the unit size if the mode specifies unit structuring. For record-structuring, only strings produced previously using DEFINE.STRING.OUTPUT should be used.

7) DEFINE.STRING.OUTPUT(I,[C],I,I)I DSO

This procedure enables a vector of bytes specified by P2 to be assigned to output stream P1, so that subsequent output operations on this stream write to the byte vector. If P1 is negative a free stream number will be selected by the system. The stream number actually used is always returned as the integer result of the procedure. P3 gives the mode, as for DEFINE.OUTPUT, and P4 gives the unit size or maximum record size if the mode specifies unit or record structuring.

8) END.INPUT(I,I) EI

This procedure is used to release an input stream which is no longer required. P1 specifies the stream to be released.

Normally, programs which define input streams release them using END.INPUT with P2 positive. This will release the stream, unless it was last assigned using the STRO* or "current stream" form of document name (see DEFINE.INPUT), in which case the stream continues to exist. If P2 is negative, the stream will be released regardless. (Negative P2 is used when the stream is being released abnormally, as a result of an error).

9) END.OUTPUT(I,I) EO

This procedure is used to release an output stream which is no longer required. P1 specifies the stream to be released.

If P2 is positive, the final section of the output stream is dispatched to its destination in the usual way, before being released. For negative P2, the action depends upon the mode associated with the stream, and the type of stream. For a message output stream, the output will be dispatched unless the DISCARD bit in the mode (see DEFINE.OUTPUT) is set. For other stream types, the output will be discarded unless the SAVE bit in the mode is set. (Negative P2 is used when the stream is being released abnormally as a result of an error. By default, message streams are dispatched in this case, while file streams are not. The SAVE and DISCARD bits allow this default action to be overridden if necessary).

As with END.INPUT, the stream is not released for positive P2 if it was last assigned using the STRO* or "current stream" form of document name.

Examples of Use of Stream Definition Commands

Of the commands described in this section, only DEFINE.INPUT and DEFINE.OUTPUT are normally used as job control commands. INIT.IO cannot sensibly be used as it leaves all streams undefined, and so there is nowhere from which further commands could be read. CHANGE.DEST cannot be used easily as the command interpreter does not provide any facility for representing vectors. The following examples show the use of DEFINE.INPUT and DEFINE.OUTPUT in some common situations.

i) DI 0 *

This is the command used automatically at the start of a batch job, to inialise its default input stream. A message input stream is defined, with all terminators suppressed except the last. Any attempt to read a new section with no message present results in a trap.

ii) DI 0 * %100

This is used automatically at the start of an interactive job to initialise its default input stream. The difference between this and (i) is that in this case, reading a new section with no message present results in a prompt being output on stream 0, and the process being halted until a message arrives.

iii) DI 1 MFY

This assigns the file MFY to stream 1. This is the kind of command normally used to set up input streams prior to entering a user program.

iv) DI 2 MF00 %100

This assigns the sequence of files MF00, MF01, ... to input stream 2. A trap will be generated if, on trying to read a new section, the required file is found not to exist.

v) DI 5

This is similar to case (iii) above, but the current file is assigned to stream 5.

vi) DO 0 LPT*

This is the command used automatically at the start of a batch job, to initialise its default output stream. A long message stream of one section, with an installation-defined size limit, and directed at the lineprinter control process, is defined.

vii) DO 0 0 %8100

This is the command used automatically at the start of an interactive job, to initialise its default output stream. A short message stream is defined, synchronised with the output so that the process awaits printing of each output section before continuing. Each section of output is sent as a reply to the process from which the current section of input stream 0 was received (i.e. the interactive terminal).

viii) DO 1 OFILE

This assigns the file OFILE to output stream 1 - i.e. on a 'BREAK.OUTPUT', the stream will be filed as OFILE. Only one section, of an installation-defined length (usually "infinite"), will be permitted.

ix) DO 2 OFILE01 0 50 10

This defines an output stream which will produce the sequence of files OFILE01, OFILE02, ..., OFILE10. The stream will be broken into sections every 50 K bytes unless explicitly broken earlier than this by the program.

x) DO 5 LPT* 0 10

This defines an output stream, directed at the lineprinter, consisting of one section (actually, an installation defined default which is normally one) of at most 10 K bytes.

Examples of Automatic Stream Definition

Many of the examples given in Chapter 2 were in fact special cases of this facility.

i) NEW FILEX
FILEX is an automatically assigned output stream, directed at the file specified.

ii) ALGOL FILEX
An automatically assigned input stream, causes the source program on FILEX to be compiled.

iii) EDIT FILEX FILEY
Automatically assigned input and output streams, FILEX is edited to produce FILEY.

iv) FORTRAN
This causes the source program on the current file to be compiled if one exists, otherwise the source program is taken from the currently selected input stream.

v) EDIT
The current file (if one exists) is edited, to produce a new current file. In this case, if no current file exists, a fault is generated.

vi) LF FILEX LPT*
The input stream is from the FILEX, the output stream is set up, with default parameters, to the lineprinter.

vii) LF FILEX STR1*
In this case, the output stream is defined to be stream 1, which must have been set up previously by a DEFINE.OUTPUT command. This is useful (a) if it is required to use an output stream with parameters other than the defaults, for example one with more than one section, and (b) if it is required to append the listing to the end of a stream which has already some output on it.

viii) LF
The current file is listed on the currently-selected output stream (c.f. v above). A fault is generated if no current file exists.

CHAPTER 4. BASIC INPUT/OUTPUT OPERATIONS

The procedures in this Chapter are intended mainly for use in programs, rather than as job control commands, and enable a program to perform the following functions

i) Select a particular stream to be used in all future input or output operations until another stream is selected, and discover which stream is currently selected.

ii) Explicitly break an output stream into sections before the size limit for a section is reached.

iii) Discover the mode, and other information, associated with a stream.

iv) Perform actual input/output operations.

The basic input/output operations described in section 4.3 all operate on the currently selected input or output stream, as specified by the SELECT.INPUT and SELECT.OUTPUT procedures.

4.1 PROCEDURES FOR STREAM SELECTION, ENQUIRY ETC

1) SELECT.INPUT(I) SI

This procedure selects input stream P1 as the current input stream. All subsequent input operations until the next call of SELECT.INPUT will read from this stream. A fault is indicated if the stream is not defined at the time of selection.

On selecting a new input stream, the current position in the currently selected stream is remembered, so that upon re-selection input can resume from the present position.

2) SELECT.HEADER()

In the case of an input section which is a long message, this procedure causes subsequent input on this section to be from the header. For short messages, input re-commences from the start of the section. On switching to a new section, the header or main document is selected according to the mode of the input stream (see 3.3.1 DEFINE.INPUT).

3) SELECT.DOC()

This procedure selects the main document of a long message. For a short message, the action is identical to SELECT.HEADER. On switching to a new section, the header or main document is selected according to the mode of the input stream (see 3.3.1 DEFINE.INPUT).

4) BREAK.INPUT(I) BI

This procedure advances input stream P1 explicitly to the next section. If P1 is negative, the current stream is advanced.

5) CURRENT.INPUT()I

This procedure returns as its result the stream number of the currently selected input stream.

6) I.MODE()I

This procedure returns the mode bits (see 3.3.1, DEFINE.INPUT) for the current input stream. The file name (if any) associated with this input stream is returned in the global variable PWW1, and the user name in PWW2.

7) I.SEG()I

This procedure returns the segment number (if any) for the current input stream. A negative result implies there is no segment - i.e. the current section is a short message.

8) I.POS()I

This procedure returns as its 32-bit result the current position in the current input stream.

For character streams, the position is returned as a page and line number, with page occupying the most significant 16 bits and line the least significant.

For binary streams, the byte position within the stream is returned.

For unit and record streams, the unit or record number is returned.

9) I.BPOS()I

This procedure returns a 32-bit index into the current input stream. This may be used subsequently as a parameter to SET.I.BPOS.

10) I.SOURCE([I])

This procedure yields the identification of the process which sent the current section of the current input stream, in a vector of four I elements supplied as a parameter. A trap is entered if the input stream is not a message stream.

```
P1[0] := system process number (SPN)
P1[1] := process identifier (PID)
P1[2] := channel to which replies are to be sent
P1[3] := message sequence number.
```

This information is in a suitable form for use as a parameter to CHANGE.DEST or SEND.MESSAGE.

11) I.ENQ()I

This procedure allows a process to enquire whether any input is currently available at the current input stream. The integer result is interpreted as follows:

```
 ___________________________________________
|                                  |d|c|b|a|
      (d)    END OF INPUT          -| | | |
      (c)    END OF AVAILABLE INPUT---| | |
      (b)    END OF SECTION        -----| |
      (a)    END OF UNIT/RECORD    -------|
```

Notes:

(a) is set when reading binary or character information from a unit or record-structured stream, if the last item in the current unit or record has been reached. Any further attempt at sequential binary/character input before selecting a new unit or record will generate a fault.

(b) is set when the last item (character, binary element, unit or record) of a section has been read.

(c) is set when the end of a section has been reached and no further sections are available. For an online message stream (see 3.4.4), subsequent attempts to input an item sequentially from the stream will cause the process to be halted. In all other cases, attempts to perform further input will be faulted.

(d) is set when the last item (character, binary element, unit or record) of a stream has been read. A subsequent attempt to input an item sequentially from the stream will generate a fault.

With (b), (c) and (d) for unit and record structured streams, character and binary input within the current unit or record is still possible.

12) SELECT.OUTPUT(I) SO

This procedure selects output stream P1 as the current output stream. All subsequent output operations until the next call of SELECT.OUTPUT will output to this stream. A fault is indicated if the stream is not defined at the time of selection.

On selecting a new output stream, the current position in the currently selected stream is remembered, so that upon re-selection output can resume from the present position.

13) CURRENT.OUTPUT()I

This procedure returns as its result the stream number of the currently selected output stream.

14) O.MODE()I

This procedure returns the mode bits (see 3.3.1, DEFINE.OUTPUT) for the current output stream. The document name associated with the stream is returned in the global variable PWW1, and the machine name (if relevant) in PWW2.

15) O.SEG()I

This procedure returns the segment number associated with the current output stream. A negative result implies that there is no segment - i.e. the stream is a short message or string stream.

16) O.POS()I

This procedure returnes as its 32-bit result the current position in the current output stream. The result is encoded in the same way as for I.POS.

17) O.BPOS()I

This procedure returns a 32-bit index into the current output stream. This may be used subsequently as a parameter to SET.O.BPOS.

18) BREAK.OUTPUT(I)   BO

This procedure terminates the current section of the specified output stream, and dispatches it to its destination. P1 specifies the output stream to be broken, where -1 indicates that the currently selected stream is to be broken. In the case of breaking output on the last section of a stream (as determined by its section count), further attempts to output to it will generate a trap (OUTPUT EXCEEDED). BREAK.OUTPUT has no effect on an undefined stream.

19) OUT.HDR([C])

This procedure replaces the header of the current output stream by the string in P1. It has no effect in the case of an unbuffered output stream.

4.2 BASIC INPUT/OUTPUT OPERATIONS

The procedures described in this section provide the basic input/output operations, in terms of which all other higher level input/output is implemented.

As explained in section 3.2.2, the operations fall into four main categories: character, binary, unit and record-organised input/output. In each case there is the notion of an address or position within the stream: for character streams this is a page and line number; for binary streams, a byte position; for unit and record-organised streams, a unit or record number. The position at any time may be found by calling I.POS or O.POS. Sequential input/output operations always advance to the next position, while random operations are achieved by first moving the pointer to a specified position and then performing the corresponding sequential operations. Random operations for character and record input/output are only implemented in a rudimentary form, using a byte position as the address.

The positioning operations for use with random input/output are

1) SET.I.BPOS(I,I)

This sets the current position for the current input stream. The parameter P1 is interpreted as a byte position, as returned by I.BPOS, for all stream types. P2 gives the logical position, as returned by I.POS. It is intended for use by higher level library modules, in implementing indexed file organisations.

2) SET.O.BPOS(I,I)

This sets the current position in the current output stream. The parameter P1 is interpreted as a byte position, as returned by O.BPOS, for all stream types. P2 gives the logical position, as returned by O.POS. It is intended for use by higher level library modules, in implementing indexed file organisations.

In the case of unit and record structured streams, there are two possible ways of accessing the current unit or record. One is simply to use the ordinary sequential character or binary input/output operations within the current unit or record. Alternatively, the unit or record may be accessed directly as a vector of bytes. The following two procedures will return a byte descriptor to the current unit or record for this purpose.

3) I.VEC()[C]

This procedure returns a descriptor into the current input stream. For unit and record structured streams, the descriptor describes the current unit or record as a vector of bytes; for other stream types it will describe an arbitrary portion of the stream.

4) O.VEC()[C]

This procedure returns a descriptor into the current output stream. For unit and record structured streams, the descriptor describes the current unit or record as a vector of bytes; for other stream types it will describe an arbitrary portion of the stream.

4.2.1 CHARACTER INPUT/OUTPUT PROCEDURES

A character stream consists of a sequence of characters, structured internally into pages and lines by means of the characters formfeed and linefeed. The first page of a file is page 1, and the first line of a page is line 1. Thereafter pages and lines are normally numbered sequentially, with the end of each page marked by a formfeed and the end of each line by linefeed. As explained in section 3.2.2, the operations described in this section and the higher level operations based on them may be used on character streams, and also within the current unit or record of a unit or record-structured stream.

The main operations are IN.CH and OUT.CH. Two additional procedures are NEXT.CH, which inspects the next character on the stream without actually advancing the pointer, and IN.BACKSPACE which provides limited facilities for backspacing in the input stream. There are also procedures to enable character streams to be processed a line at a time.

1) IN.CH()I

This procedure yields as its integer result the next character on the currently selected input stream. An error is generated if the stream is undefined, or the last character of the stream (or of the current unit or record, if appropriate) has already been read.

After reading a formfeed or linefeed character, the current position (as returned by I.POS) is not changed until the first character of the next line is read.

2) NEXT.CH()I

This procedure has the same effect as IN.CH, except that the input pointer is not advanced. Thus many consecutive calls to NEXT.CH will yield the same result. The next call to IN.CH will also yield the same result.

3) IN.BACKSPACE(I)

This procedure backspaces the input pointer by an amount specified by P1. If P1 is negative, the pointer is moved to the start of the current line (or unit or record when reading in unit or record-organised streams). This can still be used even after a linefeed or formfeed has been read. For positive P1, the pointer is moved back P1 characters, or to the start of the current section, whichever is nearer.

Note that after backspacing beyond a formfeed, the correct page and line numbers will no longer be returned by I.POS.

4) SKIP.LINE()

This procedure advances the current input stream to the start of the next line, setting the page and line numbers to the correct values for the new line.

If the pointer for the stream is currently positioned at the end of a line - i.e. a linefeed or formfeed has been read, but the line number has not yet been advanced - the line number will be advanced without moving the pointer.

5) IN.LINE([C])I

This procedure copies a line of input from the current input stream into the buffer specified by P1, and returns as its result the number of characters in the line. The linefeed or formfeed which terminates the line is copied into the buffer, but is not included in the character count.

If the pointer for the stream is currently positioned in the middle of a line - i.e. the linefeed or formfeed has not yet been read - the remaining characters (if any) in the line are copied.

6) OUT.CH(I)

This procedure outputs its parameter as the next character of the currently selected output stream. If the current section buffer is full, BREAK.OUTPUT is called first.

7) OUT.BACKSPACE(I)

This procedure backspaces the output pointer by an amount specified by P1. If P1 is negative, the pointer is worked to the start of the current line. For positive P1 the pointer is moved back P1 characters, or to the start of the current section, whichever is nearer.

8) OUT.LINE([C],I)

This procedure outputs a line, contained in the character string P1, to the currently selected output stream.

P3 specifies the carriage control to be used, and is to be interpreted as follows:

0 - overprint previous line (i.e. output carriage return before printing).
1 - print on next line (i.e. output one linefeed before printing).
2-63 - leave specified number (1-62) of blank lines before printing (i.e. output 2-63 linefeeds first).
64 - print at start of next page (i.e. output one formfeed first.
65 - output one linefeed after printing.
66-127 - output 2-63 linefeeds after printing.
128 - output formfeed after printing.

4.2.2 BINARY INPUT/OUTPUT PROCEDURES

A binary stream consists of a sequence of bytes of data, with no implied interpretation or structuring of the data. The position within a binary stream is specified as a byte number, starting from zero. As explained in section 3.2.2, the operations described in this section may sensibly be used on binary streams, and also within the current unit or record of a unit or record-structured stream.

Binary streams may be read and written, sequentially or randomly, in units of one byte or one single-length word (I). There will usually be a machine-dependent restriction, that a word may only be read or written on a full word boundary. Thus the only machine independent use of binary streams is always to read and write the same sized units on a given stream.

The sequential operations for binary streams allow bytes and single-length words (I) to be read and written sequentially. Random-access operations involve first calling SET.I.BPOS or SET.O.BPOS, then performing the appropriate sequential operation. The sequential operations available are:

1) IN.BIN.B()I	Read binary (byte)
2) IN.BIN.S()I	Read binary (single length)
3) OUT.BIN.B(I)	Write binary (byte)
4) OUT.BIN.S(I)	Write binary (single-length)

4.2.3 UNIT-STRUCTURED INPUT/OUTPUT PROCEDURES

A unit-structured stream consists of data structured into equal-sized records, called *units*. The position within such a stream is given by the unit number (assigned sequentially from zero), and the byte or character position within the unit. The procedures described in this section should *only* be used on unit-structured streams.

Unit-structured streams may be read and written both randomly and sequentially. Random access is achieved by calling SET.I.BPOS or SET.O.BPOS, and then performing the appropriate sequential input/output operations. If preferred, the unit may be accessed directly using I.VEC or O.VEC (see 4.2).

The procedures IN.UNIT and OUT.UNIT operate sequentially on the currently selected stream.

The standard sequence for input from/output to a unit is:

INPUT
SET.I.BPOS (if random)
IN.UNIT

sequential character/binary input

OUTPUT
SET.O.BPOS (if random)
sequential character/binary output
OUT.UNIT

In both cases, the sequential character/binary operations may be replaced by a call to I/O.VEC and direct access to the unit. If the SET.I.BPOS/SET.O.BPOS is omitted, sequential accessing results.

1) IN.UNIT()

This procedure selects a particular unit in the currently selected input stream. Subsequent sequential character or binary operations will operate within this unit, and a call to IN.VEC will return a descriptor to this unit.

If SET.I.BPOS has been used, the unit selected is the one which was specified there; otherwise the stream is advanced to the next unit in sequence.

2) OUT.UNIT()

This procedure outputs the current unit on the current stream, and advances the stream to the next unit.

4.2.4 RECORD-STRUCTURE INPUT/OUTPUT PROCEDURES

A record-structured stream consists of data structured into records which need not all be of the same size. The position within such a stream is given by the record number (assigned sequentially from zero) and the byte or character position within the record. The procedures described in this section should only be used on record-structured streams.

Record-structured streams may only be read and written sequentially (though a restricted form of random I/O can be implemented using SET.I.BPOS and SET.O.BPOS). The actual procedures have the effect of advancing the stream to the next record, so that subsequent character or binary operations operate within that record. Alternatively, the record may be excessed directly, using I.VEC or O.VEC.

The procedures IN.REC and OUT.REC operate sequentially on the currently selected stream. OUT.REC has an integer parameter giving the number of bytes in the record to be written. This is only used if the records are written directly, using O.VEC. In the case where the records have been written using character or binary operations, the parameter should be set to -1, in which case the size of the record is computed from the current position of the appropriate stream pointer.

The standard sequence for input from/output to a record is:

<u>INPUT</u>	<u>OUTPUT</u>
SET.I.BPOS(for "random")	SET.O.BPOS(for "random")
IN.REC	sequential character/binary output
sequential character/binary input	OUT.REC

In both cases the sequential character/binary operations may be replaced by a call to I/O.VEC and direct access to the record.

For IO streams (see 3.2.5), care must be taken when outputting records, to ensure that the size of an output record is the same as the size of the corresponding input record.

1) IN.REC()

This procedure selects the next record in the currently selected input stream. Subsequent sequential character or binary operations will operate within this record, and a call to IN.VEC will return a descriptor to this record.

2) OUT.REC(I)

This procedure outputs the current record on the currently selected stream, and advances the stream to the next record, P1 gives the size of the record. If P1 is negative, the size is to be computed from the number of sequential character/binary items output.

4.3 OTHER (CHARACTER STREAM) INPUT/OUTPUT PROCEDURES

For the majority of user's, the input/output interface is defined by the programming languages they use, and is implemented in terms of the basic procedures described in section 4.2. However, there are some further procedures available in the library for doing common character input/output operations (such as reading and printing decimal integers). These are used in the implementation of the basic system, and in effect form the input/output interface of the system programming languages, but they may also be used by programs written in other languages.

4.3.1 CHARACTER STREAM INPUT PROCEDURES

1) IN.I()I

This procedure reads a (possibly signed) decimal integer from the currently selected input stream, returning the value as its result. A trap is forced if the next nonblank character is not a decimal digit, +, or -.

2) IN.OCT()I

This procedure reads an octal integer from the currently selected input stream, returning the value as its result. A trap is forced if the next nonblank character is not an octal digit.

3) IN.HEX()II

This procedure reads a hexadecimal number from the currently selected input stream, returning a double length value as its result. A trap is forced if the next nonblank character is not a decimal digit, A, B, C, D, E or F.

4) IN.C.LIT()II

This procedure reads a string of characters enclosed in double quote symbols from the currently selected input stream, returning the string as a packed, right-justified *II* value. If too many characters are given, the required number are taken from the end of the string. Non-printing characters can be represented by using their ISO character codes written as two hexadecimal digits and enclosed between exclamation marks, e.g. "ABC", "A VERY LONG STRING",

"!OC!PAGE3". A trap is forced if the next character is not ".

5) IN.NAME()II

This procedure is similar to IN.C.LIT except that the string should not be enclosed in quotes, preceding blank lines and spaces are ignored, and the string terminates on reading either a space or a newline. (This procedure is commonly used in the system for reading filenames, etc.).

6) IN.C.STR([C])I

This procedure reads a character string enclosed in double quotes from the currently selected input stream, in the same format as for IN.C.LIT above. The string is placed in the byte vector described by P1, and its size (number of characters) is returned as an integer result. If the size of the byte vector is too small to accommodate all the characters given, any remaining characters are ignored up to the end of the string.

7) IN.STR([C])I

This procedure is similar to IN.C.STR except that the string should not be in quotes, preceding blank lines and spaces are ignored, and the string terminates on reading either a space or a newline. (This procedure is used by the job control interpreter to read string parameters). Notice that spaces within the string must be represented by their hexadecimal character codes, e.g. A!20!STRING!20!WITH!20!SPACES.

4.3.2 CHARACTER STREAM OUTPUT PROCEDURES

1) SPACES(I) SP

This procedure outputs the number of spaces specified by P1 to the currently selected output stream.

2) NEW.LINES(I) NL

This procedure outputs the number of newlines specified by P1 to the currently selected output stream. If P1 is zero, and the previous character on this stream is newline, nothing is output; otherwise a single newline is output.

For short message streams (e.g. online output), BREAK.OUTPUT is called to end the current section on each call to NEW.LINES.

For long message streams, BREAK.OUTPUT is only called when the number of lines in the section reaches the specified section size limit (see 3.2.3).

3) CAPTION([C]) CAP

This copies the string of characters in the byte vector P1 to the currently selected output stream.

4) OUT.I(I,I)

This procedure prints the integer P1 as a decimal integer on the currently selected output stream. P2 specifies the required field width. If P2 = 0, the number is printed left-justified. Otherwise, it is printed (if possible) in a field width of P2 characters, preceeded by a space for positive numbers, a minus sign for negative numbers (P2+1 characters in total). If the number requires more than P2 digits, it will overflow the specified field width.

5) OUT.OCT(I)

This procedure prints the parameter P1 as an 11-digit octal number on the currently selected output stream.

6) OUT.HEX(I)

This procedure prints the parameter P1 as an 8-digit hexadecimal number on the currently selected output stream.

7) OUT.TIME()

This procedure prints the time, in the format HH:MM:SS (hours, minutes, seconds) on the currently selected output stream.

8) OUT.DATE()

This procedure prints the date, in the format DD:MM:YY (day, month, year) on the currently selected output stream.

9) OUT.TD(II,I)

This procedure prints the time or date, extracted from its parameter P1. This parameter is assumed to be a 32 bit integer, as returned by the system procedure TIME.AND.DATE (Chapter 22). If P2 is zero, the time is printed in the form HRS HRS:MINS MINS:SECS SECS. If P2 is non zero, the date is printed in the form DAY DAY:MONTH MONTH:YEAR YEAR.

10) OUT.LINE.NO(I)

This procedure prints the packed page/line number specified by P1 on the current output stream. The page and line numbers are printed in decimal, separated by '.', with a total field width of 10 characters. P1 should be in the form returned by I.POS (4.1).

11) OUT.NAME(II)

This procedure outputs its parameter as 8 characters, left justified, with spaces to the right replacing any leading null characters.

12) OUT.FN([C])

This procedure is used by compilers etc., to print out 'filename time date' at the head of compilations, file listings, etc. P1 is the filename to be printed. If it is zero, the filename <CFILE> will be printed.

13) OUT.STACK(I,I)   OS

This produces a hexadecimal dump on the currently selected output stream of the area between byte addresses P1 and P2. Any number of consecutive identical lines are replaced by one copy of the line followed by a single blank line.

14) ECHO.LINE()

This causes the current line of input to be printed on the current output stream if the output stream is offline. Otherwise, it has no effect.

15) PROMPT([C])

This procedure resets the system prompt message, so that the specified string (P1) is used the next time the system prompts an online user for input. The prompt string will be reset to '**' by the job control interpreter next time it regains control. If the parameter is zero, prompting is supressed completely.

CHAPTER 5. FILES AND EDITING

5.1 GENERAL DESCRIPTION OF THE FILE SYSTEM

The file system provides users with the means of retaining information inside the system, and of accessing and altering this information from within jobs. Often, though not always, the files are created by programs or system commands, using the input/output facilities described in Chapters 3 and 4.

Each user of the system has a file directory, which lists the files currently owned. Normally, all file accesses in a job refer to the directory of the user who owns the job; however, facilities are also provided for accessing the files of other users with their permission. A single file may contain up to 256 K bytes of information, but individual users are restricted both in the number of files they may own and in the total amount of space their files may occupy. If a user exceeds either of these limits, further attempts to access the files will be prevented until the usage is brought back within the allocated limits - either by deleting files or by increasing the user's allocation.

A user does not normally need to be aware of the exact location of his files, as this is managed automatically by the system. There is one aspect of this, however, which may concern the user. The file system can extend across up to three levels of storage, and files are automatically transferred by the system between the three levels as necessary. Files which are currently in use or have recently been accessed reside in the <u>local</u> filestore. Files which have not been used for some time may be automatically offloaded to a large capacity file buffer, from which they may be further removed to archive storage on removable storage media (e.g. magnetic tapes). The user need not be aware of this movement, since files will be automatically retrieved into the local filestore when they are accessed, but retrieving a file from the archives may involve an appreciable delay. Also, commands are provided for the user explicitly to request file offloading and onloading, for security reasons. This mechanism is only provided on machines with a suitable configuration.

5.2 SHARING FILES

Two methods are provided whereby users can access files belonging to others. In the first, the OPEN.DIR command is used to select another user's directory for subsequent file operations. This allows the same access to all of the files as the owner of the directory, and requires that the owner's password be given as authorisation.

The second method allows more selective sharing of files, without the requirement to know the owner's password. Any user may permit others selectively to share his files by calling on the PERMITT command, giving the access permission granted to each user. Provided that the appropriate permission has been granted, a user may access another user's file by specifying "filename/username", with no intervening spaces, whenever a "filename" parameter to a command is required.

5.2.1 Network File Systems (not applicable to stand alone systems)

Each machine in the MU6 network has its own independent file system. However, the file commands are such that users are able to open directories and retrieve copies of files from other machines. This involves passing messages between the different file systems. To specify the necessary information about the required directory in another machine, a command REMOTE.DIR is provided. This assigns a new name for the required directory, which may subsequently be used in the file commands to access the remote files.

It is essential that users have been granted the necessary permission to access the files in the remote machines.

5.3 FILE SYSTEM COMMANDS

1) OPEN.DIR(II,II)

This command enables a file directory to be selected for use in subsequent file operations. P1 gives the username of the user whose directory is to be selected and P2 gives the password.

2) OPEN.FILE(II,II,I,I)

This command opens a file P1 into the virtual store of the current process. P2 is the username of the file's owner, with a default of zero implying the currently selected directory. P3 and P4 specify a segment and access permission for the file. If the segment number is negative, a free one will be allocated by the system. Any access permission is allowed for files in the current directory, but for files borrowed via the PERMIT facility, the calling process will be faulted if the specified access exceeds that stated in the PERMIT file. PW1 returns the number of the segment holding the file, and PW2 and P3 the size and status of the segments respectively.

3) FILE(II,II,I)

This command preserves a segment (P3) as a file of name P1. P2 is the name of the user who will subsequently own the file, with a zero default implying the current directory. The segment remains in the process' virtual store after a file operation.

4) DELETE.FILE(II,II)

This command deletes the file P1 in directory P2. If the file is not in the current directory, the user must have permission to delete the file in PERMIT.

5) RENAME.FILE(II,II,II)

This command is used to rename the file P1 in directory P2. P3 gives the new file name. If the file is not in the current directory, the user must have permission to rename the files in PERMIT.

6) SECURE.FILE(II,II)

This command causes the named file to be copied to the offload buffer, thus creating a secure copy of it. The local copy of the file is retained. P1 gives the name of the file and P2 the directory name. Update permission is required to secure a file in another user's directory.

7) BACKUP.FILE(II,II)

This command deletes the local copy of a file, so that on next accessing the file, the most recently offloaded copy (if any) will be obtained. This may be the version last copied using the SECURE.FILE command, or it may be a later version which has been offloaded automatically by the system. P1 gives the name of the file and P2 the directory name. Update permission is required to secure a file in another user's directory.

8) CATALOGUE.FILES()

This command creates a copy of the currently selected file directory in a newly created segment. PW2 returns the segment number. It contains directory entries of the form:

FILE NAME(64 bits)
Status
Size in bytes
Create time
Create date

Each entry occupies 32 bytes. PW1 holds the number of entries in the directory, PW3 is the maximum number of files permitted, PW4 is the amount of file store used (in K bytes) and PW5 is the maximum amout of filestore allowed to the user.

9) READ.FILE.STATUS(II,II)

This command reads the directory information associated with the file name P1 in directory P2. PW1 returns the status information in the form:

```
 ________________________
 |      |   |   |   |   |
 |______|___|___|___|___|
            |   |   |   |
            |   |   |    ----- local copy exists
            |   |    ------offloaded copy exists
            |    ----- file open
             ----- file open with exclusive access
```

PW2 holds the size of the file in bytes, and PW3 and PW4 returns the create time and date respectively.

10) PERMIT(II,II,II)

This command may be called by a user to assign a set of access rights to one of his files for another user. P1 specifies the filename and P2 the name of the user to be granted permission. The option "ALL" is available for both parameters, to allow a user access to all the files in a directory, or allow all users access to a file.

The third parameter gives the permissions assigned, and is specified as a combination of the letters:

X	meaning permission to open with execute access
W	meaning permission to open with write access
R	meaning permission to open with read access
E	meaning permission to open with exclusive access
C	meaning permission to change access when open
U	meaning permission to update the file
D	meaning permission to delete the file
N	meaning permission to rename the file.

The letters may appear in any order.

To remove access permission rights, the PERMIT command should be called with P3 zero.

11) GET.PERMIT()

This command creates a copy of the permissions associated with the currently selected file directory. A segment is created for this purpose, and the segment number is returned in PW1. The permission entries are of the form:

FILENAME	(64 bits)
USERNAME	(64 bits)
ACCESS	(32 bits)

The access is bit significant, and has the format

```
 _______________________
|______|_|_|_|_|_|_|_|_|
        | | | | | | | |
        | | | | | | |  -- X
        | | | | | |  ---- W
        | | | | |  ------ R
        | | | |  -------- E
        | | |  ---------- C
        | |  ------------ U
        |  -------------- D
         ---------------- N
```

The number of entries in the segment is returned in PW2.

12) REMOTE.DIR(II,II,II,II,II)

This command provides a mechanism for accessing files from another machine. It allows a user to define a new directory name, P1, which may be used as the user name parameter of subsequent file commands.

The remaining parameters provide information necessary to identify the remote machine and the users in it. P2 and P3 give the user name and password of the current user in the remote machine. P4 gives the machine name. If this is zero, the current machine will be assumed. P5 is the name of the user whose directory is to be accessed in the remote machine. Before file operations can take place, it is necessary for the user to have given the required permissions to the user P2. If the P5 parameter is zero, the directory for P2 will be accessed i.e. the current users directory but in the remote machine.

5.4 FILE EDITING

5.4.1 GENERAL DESCRIPTION OF THE EDITOR

The editor is a program which enables users to make alterations to files of text. It is invoked by the command

EDIT(II,II)   ED

The two parameters of this command are filenames (or, more accurately, document names - see Chapter 3). The first is the input file, and the second the output file, and the action of the editor is to make an altered copy of the input file on the output file. The two filenames may be the same, in which case the input file is overwritten at the end of the edit by the output. In general, the two names may take any of the forms described for input and output documents in Chapter 3, except that unbuffered (online) documents may not be used.

The use of the editor can be defined in terms of three operations, namely

1. Copying information unchanged from the input file output file.
2. Skipping over information in the input file, without copying to the output file.
3. Inserting new information into the output file.

So, for example, to change a file saying

```
1.  'BEGIN'
2.  'REAL' X, Y;
3.  X := REAL
4.  PRINT (X+Y,3,2)
5.  'END'
```

To one in which the third line says

```
3.  Y := READ;
```

We might copy lines 1-2, skip line 3, insert the new line 3 and then copy the rest of the file. The editor provides a variety of commands for copying, skipping and inserting, allowing different ways of specifying exactly what is to be copied, skipped or inserted.

5.4.2 ONLINE OPERATION

If the editor's commands are taken from an online stream, the editor will operate in online mode. In this mode, fault messages and other monitoring will normally be returned to the online console. In the event of a fault the editor prints a fault message and then ignores any further commands on the same line. If the command in which the fault occurred was an editing command (S, C, A, B, D) then the input and output pointers are restored to their values at the start of the command.

5.4.3 OFFLINE OPERATION

In offline mode, as soon as a fault occurs the editor returns control to the calling program with the status word (PWO) set to -1. This will normally have the effect of terminating the job. No further editing commands are read, since there is a high probability of interpreting inserted data etc. as commands.

5.4.4 COMMANDS AND PARAMETERS

Operation of the editor is controlled by means of a sequence of commands on the command input stream. Each command is identified by a single letter, which may be followed by a parameter. Any number of commands may be placed on a line; blank lines and spaces occurring between commands are ignored. Edit commands are accepted in upper or lower case.

Most parameters are straightforward (e.g., an integer) and require no special explanation. However, a number of commands have as their parameter a string. A string is simply a sequence of characters betweeen a pair of string delimiters. Any single non-alphanumeric character (except '+') which does not appear in the sequence of characters may be used as a string delimiter. Thus, any of the following are legal strings:-

```
1.  'ABC DEF'
2.  /'BEGIN'/
3.  $'BEGIN'
     'INTEGER' I, J, K;
    I := 1/2;
    $
```

Sometimes it may be necessary to include as part of a string a character which cannot be typed on the input device, (for example a newpage character). This may be achieved by typing the code for the required character as two hexadecimal digits enclosed between two string delimiters. Thus, for example using the ISO character codes:-

1. !'BEGIN'!09!'INTEGER' I,J,;! means 'BEGIN' followed by a tab followed by 'INTEGER' I,J;,

2. !!OC!TITLE! represents a newpage character followed by TITLE.

3. //OA//OA// represents two consecutive newline symbols.

<u>Note</u> that this will only work for characters with codes in the range 00 - 9F. Certain characters are required so frequently that a special representation exists within strings for them. This consists of a single letter enclosed between a pair of string delimiters. The characters for which a special representation exists are:-

Newpage	(P)	e.g.,	//P/TITLES/
Newline	(L)	e.g.,	//L//L//
End of File	(F)	e.g.,	//F//

N.B. Users should be very careful in using the hexadecimal representations, particularly in insertions, as it is possible to introduce non-standard non-printing characters into the output file. Also there may be unexpected effects when using line-editing commands (see later).

5.4.5 BRIEF SUMMARY OF AVAILABLE COMMANDS

A brief description of the available commands is given below. They will then be described individually in greater detail. There are 9 commands altogether, and they are grouped for ease of description into 3 categories. Commands from all 3 categories may, however, be freely interspersed.

A. Line-editing Commands

1. S (SKIP)Skips to the start of a specified line.
2. C (COPY)Copies to the start of a specified line.
3. I (INSERT)Inserts a string into the output stream.

B. Context-editing Commands

1. B (BEFORE)Copies up to the start of the next occurrence of a specified string.
2. A (AFTER)Copies up to the end of the next occurrence of a specified string.
3. D (DELETE)Copies up to the start of the next occurrence of a specified string, then skips to the end of the string (thus deleting it).
4. I (INSERT)Inserts a string into the output stream (exactly as for line-editing).

C. Other Commands

1. W (WINDOW)Causes a section of the file being edited to be listed on the monitor stream.
2. R (RESTORE)Causes the input and output pointers to be restored to the values they had before the previous line or context editing command.
3. E (EXIT)Copies all remaining characters from the input stream to the output stream, and returns control from the editor.
4. M (MERGE)Alters the editors input file.
5. Q (QUIT)Abandons the edit and returns from the editor without updating the output file.

5.4.6 LINE EDITING COMMANDS (S, C, I)

These commands are used when whole lines are to be deleted, inserted or replaced. The two positioning commands (SKIP, COPY) always position the input pointer INP at the start of some specified line. As explained in Section 1, SKIP merely advances the input pointer without copying any information to the output stream. COPY advances the input pointer, copying all characters it passes into the output stream. The INSERT command is not specifically a line-editing command, as the same command is used in both line and context editing. It is described separately in this section as the method of use for line editing is slightly different.

For the purpose of the line-editing commands (and also to enable compilers, etc., to identify a particular line when monitoring faults) each line in a stream is identified by two 'co-ordinates', its <u>page number</u> and its <u>line number</u>. The first page of a file is page 1; thereafter pages are numbered sequentially. The end of the page is marked by the presence of a formfeed (FF) character in the file. The lines within a page are similarly numbered sequentially from 1, the end of a line being marked by a linefeed (LF) character.

The SKIP and COPY commands enable the line required to be specified in any of a number of ways, as follows, (NOTE all positions are positions in the input stream, the output stream position being specified implicitly).

1. As an absolute page and line number, e.g., S2.3 meaning skip to line 3 of page 2.

2. As a line number only, if the line required is on the current page, e.g., C19 meaning copy to line 19 on the current page.

3. As a relative page number, e.g., S + 5, meaning skip to the top (i.e., line 1) of the page numbered i + 5 where i is the current page number. Thus S + 1, means 'skip to top of next page'. (Effectivaly, skip over 5 pages including the current one).

4. As a relative line number, e.g., S + 20 meaning skip to the start of the line numbered j + 20 where j is the current line number. (Effectively, skip over 20 lines).

5. As a string, e.g., S'BEGIN' meaning skip to the start of the next line which contains the string BEGIN.

6. The letter 'F', (e.g. CF) meaning copy (or skip) to the end of the file. Because this is a dangerous command, the editor will always request verification before proceeeding, by printing 'REALLY?'. If the response to this is 'Y', the editor proceeds; otherwise the command is abandoned as faulty. Thus in offline mode, to skip to the end of the input, the sequence 'SFY' is necessary.

Note that use of control characters such as newline in strings in the SKIP and COPY commands can have rather unexpected results. For example, consider:-

```
                              C//L/'BEGIN'/
```

(or, for that matter

```
                              C/
                              'BEGIN'/  ).
```

means copy to the start of the line containing the string 'linefeed' 'BEGIN'. Thus if the file contained:-

```
                              'END' OF PROCEDURE;
                              'BEGIN'
```

the pointer would finish at the start of the 'END' line. If you don't understand this, never use control characters in S or C commands!!

Insertion

The Insert command has a string as its parameter, and inserts all characters between the string delimiters into the output stream. Since the SKIP and COPY commands always stop at the start of a line, this means that to insert a single line into the output the insert command should be used as follows:-

```
S21I/NEW LINE TO BE INSERTED
/
```

or

```
S21I
/NEW LINE TO BE INSERTED
/
```

or, of course

```
S21
I/NEW LINE TO BE INSERTED
/
etc.
```

The following:-

```
S21I/
NEW LINE TO BE INSERTED
/
```

will insert the new line with an extra blank line before it and

```
S21I/
WRONG WAY OF DOING IT/
```

will insert an extra blank line and will then simply insert the new line at the start of the existing line. This may of course be useful in some cases, e.g.,

```
S/'INTEGER'/I/
'BEGIN'/
```

will result in:

```
                    'BEGIN''INTEGER'
```

This is using the Insert command in its context-editing sense (see later), and is quite permissible provided that the user understands what he is doing.

NOTE. That when inserting new pages, the newpage character should always be placed on a new line. Thus:-

(a) To split an existing page into 2 pages such that the second starts at the old line 50.

```
C50I//P//
```

has the desired effect.

(b) To insert a completely new page betweeen the existing pages 4 and 5

```
C5.1I
/FIRST LINE OF NEW PAGE
LAST LINE OF NEW PAGE
/P//
```

Summary of Line-editing Syntax (by examples)

```
1.  C7.5        copy to page 7 line 5.
2.  S15         skip to line 15, current page.
3.     S + 3.   skip to top of next page but 2.
4.  S + 10      skip over next 10 lines of input.
5.  C'XYZ'      copy to start of next line containing XYZ.
6.  CF          copy to end of input file.
7.  SFY         skip to end of input file.
8.  I           normal single line insertion.
    /LINE TO INSERT
    /
9.  I           multiple line insertion.
    /LINE 1
    LINE 2
    ETC
    /
10. I           page insertion (if initially positioned at
    /LINE 1     start of page).
    LINE 2
    ETC
    /P//
11. I//P/LINE1     page insertion (if initially positioned at
    LINE 2         end of page).
    ETC
    /
```

5.4.7 CONTEXT-EDITING COMMANDS (B, D, A, I)

These commands may be used to edit individual characters, or strings of characters, within a line. All four commands have a single string parameter. The Insert command is actually the same command as the one used for line editing.

The 'BEFORE' command (B)

The 'BEFORE' command is basically a copy operation, copying characters from input to output until the input pointer arrives at the start of the next occurrence of a specified string. Unlike the line-editing copy operation (C), however, the input pointer is left immediately before the start of the string, not at the start of the line. For example, to correct:-

```
'BEGIN''INTER' I,J,K;
```

the commands

```
B/ER/I/EG/
```

would be used to copy to immediately before the string ER and then insert EG. In making corrections of this type, of course, it is important to ensure that the string being searched for is unique. For example, B/E/I/EG/ would have resulted in:-

```
'BEGEGIN''INTER' I,J,K;
```

The 'AFTER' command (A)

This is also a copying operation, leaving the input pointer immediately after the next occurrence of the string. Thus for example, to correct:-

```
'BEGIN''INTE I,J,K;
```

the commands

```
A/INTE/I/GER'/
```

could be used.

The 'DELETE' command (D)

The delete command is a combined copy and skip operation. Its action is to copy to the start of the specified string, and then skip to the end of the string, thus effectively deleting it from the output. This operation is especially useful for correcting spelling or typing errors, for example:-

```
'BEGONE''INTEGER' I,J,K;
```

can be corrected by

```
D/BEGONE/I/BEGIN/
```

or, provided that the string ONE does not occur before the spelling mistake, by

```
D/ONE/I/IN/
```

The 'INSERT' command (I)

Examples of the use of the insert command have already been given in the preceding section. Its action, as in the case of line editing, is simply to insert all the characters in the string into the output stream.

Strings used in context-editing commands may, of course, contain newline and newpage symbols as in the line editing commands. The most common use of the context editing commands, however, is to correct small errors within a line. Care should be taken in context editing to avoid deleting the final newline character of a page or file.

Summary of Context Editing Syntax (by example)

1. B'CONTEXT' Copy to immediately before the string CONTEXT.
2. A!END! Copy to immediately after the string END.
3. D/XYZ/ Delete the next occurrence of the string XYZ, by copying to its start and skipping to its end.
4. I/CHARS/ Insert the string CHARS into the output.

5.4.8 OTHER COMMANDS

The following commands are not actually editing commands, but perform other useful functions.

The 'WINDOW' command (W)

The window command allows sections of the edited output to be printed on the monitor stream. In offline operation, all altered pages are normally listed on the monitor stream after the commands. The appearance of any W command will inhibit this listing of altered pages.

In online mode, listing of altered pages does not occur, and the W command is the only way that the contents of the input and output streams can be inspected. In both online and offline modes, the W command may have either of the forms:-

```
W  integer e.g., W3
W
```

When used with no parameter the W command prints the current line on the monitor output stream. If the input pointer is at the start of the line, (i.e., the last character output was newline or newpage), the line is printed from the input stream. If on the other hand the input pointer is not at the start of the line, then the first part of the line is printed from the output stream, followed by the characters ~^~ indicating the current position of the pointer, followed by the rest of the line from the input stream. The line is preceded by its page and line numbers in the input stream. Thus for example, if line 4 of page 2 contained:-

```
'BEGIN''INTEGR' I,J,K;
```

then the commands

```
D/INTEGR/I/INTEGER/W
```

would cause the following to be printed

```
2.4)'BEGIN''INTEGER~^~' I,J,K;
```

The parameter N indicates that after printing the current line, the next N-1 lines of the input stream should also be printed.

The RESTORE command (R)

The restore command has no parameter and simply restores the values of the input and output pointers to their values before the last editing command, (i.e., S, C, A, B, D). This is useful if a command has been accidentally typed wrongly. For the purpose of this operation, the I command is not considered as an editing command.

The EXIT command (E)

This command copies the remaining characters in the input stream to the output stream and returns control to the program which called the editor (usually the job control interpreter).

If it is required to delete the end of a file, this must be done by using the command SF.

The MERGE command (M)

This command has a filename as its parameter, and causes the editor to switch its input to the start of the named file. Note that there is no way of returning to the previously selected file other than by a further Merge command, which will return to the start of the file.

The QUIT command (Q)

This command may be used to abandon an edit without updating the output file. It is useful as an emergency exit in the case of disastrous errors.

5.4.9 REPETITION OF COMMANDS

Any sequence of commands may be repeated a specified number of times by enclosing the sequence in brackets () with a repetition count, e.g.,

(10 S + 1 I/'COMMENT'/)	will insert the string 'COMMENT' at the start of each of the next 10 lines.
(8 D/INTER/I/INTEGER/)	will correct a mis-spelling which has been made 8 times.

Instead of the repetition count, any of the following may be used:-

L	Meaning repeat until a newline is encountered in the input stream.
P	Meaning repeat until a newpage is encountered in the input stream.
F	Meaning repeat until the end of the input stream is encountered.

When using the L and P repetition options, the pointers are left at the start of the next line or page. Thus for example:-

```
C2.1 (P D 'E' I 'X')
```

will replace all 'E's on page 2 by 'X's leaving the input and output pointers at the top of page 3 (line 1).

Note that the only effect of the (L, (P options is to restrict the range of context searches to the current line or page respectively. Thus they may not operate correctly if searches for explicit line numbers are used within the repetition.

Examples

(L D/A/I/*/)   Replaces all 'A's on the current line by '*'s.

(P D/AREA/I/AREA1/) Replaces all occurrences of the name AREA on the current page by AREA1.

(FA/MOD1/W)   Searches for all occurrences of the name MOD1 in the file, and lists each line containing such an occurrence.

NOTE. In online mode the sequence of commands to be repeated is limited to a single line. There is no such restriction in offline mode. Brackets may be nested to a maximum depth of 3 though this should rarely be necessary, e.g.,

(5 (P D/A/I/B/)) Replaces all occurrences of the letter A by B on the next 5 pages.

CHAPTER 6. DOCUMENTATION AIDS

The Documentation Aids are of two kinds; word processing and diagrammatic. Most users should find the first set of interest but some of the second set relate to the programming methodology used in producing MUSS and in this area opinions can be sharply divided.

6.1 WORD PROCESSING

This facility is provided mainly by the procedure TEXT, but a spelling checking aid SPELL augments this, and the standard editing and file facilities described in the previous chapter provide the means for manipulating the files. Text takes an encoded description of a document, and produces the required layout. It contains a facility to interface with other layout procedures, for example for tables, formulae and diagrams.

6.1.1 THE TEXT FACILITY

The procedure TEXT copies text from an input stream generating a layout suited to a specified type of printing device. It allows the user to embed "warning sequences" in the text which control the layout. Two basic device types are provided for, corresponding to 'document printers' (e.g., daisy wheel printers) and ordinary printers or terminals. All printers are assumed to have a full visual character set but the 'document printers' are also assumed to have

a variable character size
a reverse linefeed
a variable linefeed
and an underline facility.

In the specification given below actual device names are used, as the type parameter, because the document printers are not fully compatible. Normally the page and line layout is determined by the TEXT procedure, and the user is required only to provide the actual text and to indicate paragraph and section boundaries. This manual has been produced using TEXT on MU5 and serves as an example of the facility. The formal specification of TEXT is

```
          TEXT ([C],[C],II)
    where P1 specifies the input file
          P2 specifies the output file
          P3 = 0 ordinary printer (LPT)
             = DBL diablo printer
             = LPT line printer
```

It is the warning sequences and the actions they trigger that characterise the system. These are presented in three groups as the basic facilities, the layout control facilities and the special purpose layout facilities. However, most warning sequences normally start with the symbol '@'. The character immediately following the '@ ' defines the required action, some of which may require that some parameters follow.

6.1.2 BASIC FACILITIES OF TEXT

The basic mode of operation of TEXT is that it copies words from the input to the output, starting newlines (and new pages) as necessary to avoid splitting words across lines and spacing the words across a line to right justify them. A 'word' in this context is any sequence of characters in the input separated by space and/or newline characters and/or warning sequences. Warning sequences inserted in the input can modify this basic action as follows.

@ followed by a newline symbol. This causes a newline to be inserted and it inhibits the right justification of the current line.

@ followed by B. This signifies the start of a new paragraph. The current line is terminated without right justification, some blank lines are inserted to separate the paragraphs and spaces are inserted to indent the new paragraph.

@ followed by S. This indicates the start of a new section. The action is similar to @B except one more blank line is normally inserted and the following line is not indented. In fact the following line in the input is copied without any layout change to the output because it is assumed to be a section heading.

@ followed by P. This causes a new page to be started and the last line of the previous page will not be right justified.

@ followed by C or W or L. These three characters are all concerned with underlining. C causes the following character to be underlined, W the following word and L the following line. It should be noted that L relates to a line of output not a line of input. It can be used in conjunction with @S to underline a section heading in which case the line of input corresponds exactly to a line of output. It can also relate to lines which the user forcibly terminates with @ 'newline'.

@ followed by X. In addition to '@' three other symbols, namely '%', ' ' and '}' have special significance as described in 6.1.3. If text is to be processed which uses these symbols normally, other symbols can be associated with their special functions, by means of the @X command. After @X should follow a list of non-blank symbol pairs, in which the first symbol is one of @, %,   or } and the second is the symbol that is to take its place.

@ followed by F signifies the end of the text file. In order to avoid an entry to the input ended trap of the operating system '@F' should be placed at the end of all files to be processed by TEXT. If it is omitted the output text will normally still be produced.

6.1.3 LAYOUT CONTROL FACILITIES

Here we are concerned mainly with centreing, tabulating and indenting text and with superscripts and subscripts. Some effects of this kind can be obtained from the facility to change the parameters that determine the basic layout, and it is convenient to start with this. It is @ followed by V that provides the warning sequence that causes parameters to be changed. It should be followed by an arbitrary list of pairs of integers all on one line. Each integer pair gives a parameter number and its new value. The parameter numbers have the following significance

1 position of L.H. margin
2 position of R.H. margin
3 number of blank lines between paragraphs
4 number of spaces in a paragraph indent
5 number of blank lines between sections
6 number of blank lines at the head of a page
7 number of lines on a page
9 page number

Three of these require amplification because they permit special negative encodings. If a negative number is given for parameter 4 the first paragraph of each new section will not be indented. The other paragraphs will be indented by the absolute value of the number. If a negative number is given for parameter 5 each section heading will be placed on a new page. If a negative number is given as the page number, pages will not be numbered.

Any line of input can be centred on a line by preceding it with @ M. The previous line is terminated without right justification. The position of the first character on a centred line is remembered and any subsequent line of input can be similarly positioned by preceding it by @N. Again the line previous to the @N line will be terminated without right justification. In both cases the output line will correspond exactly with the input line apart from being right shifted to achieve the centralisation effect.

Tabulation is normally provided for by the '%' character. Its action is defined by the @T sequence. This serves a dual function, it defines a character which is subsequently to be treated as a 'tab' character normally '%' and it defines positions across the page for 'tab stops'. Therefore it is assumed to be followed first by a single character which becomes the tab character and then a list of integers which define the positions of tab stops counting from the L.H. margin. Whenever the tab character is used the output line will be space filled up to the next tab stop. If the output line is subsequently right justified, the extra spacing will occur only to the right of the last space inserted as a result of using a tab symbol. Obviously even this can be avoided by ending the source line with @ newline.

Indentation is provided for by a mechanism similar to the tabulation facility. The warning sequence is @I, which is similar to % but with some very significant differences. Like 'tab' when 'indent' is used the line is space filled to the next tab position. The difference is that the last indented position is remembered and if a newline is automatically generated in the output due to the current line becoming full the next line is automatically indented to the same extent, whereas in the case of tabulation the next line commences back at the margin. Any user forced newline by means of @ newline, @B etc., cancels the indentation.

Providing the output device has the facility of fractional linefeeds up and down, superscripts and subscripts are obtained by surrounding the appropriate character string with the symbols "{" meaning 'shift half a line up the page' and "}" meaning 'shift half a line down'.

6.1.4 SPECIAL LAYOUT EFFECTS

The special effects are concerned mainly with diagrams and tables contained within the text, and with special forms of document.

In the case of diagrams the simplest requirement is to leave space at an appropriate place in the text for a diagram that will be produced by other means. The warning sequence for immediate creation of space is @D which should be followed by an integer giving its size in lines. An alternative which leaves the requested amount of space at the foot of the current page if possible otherwise at the top of the next page is @E.

Table layouts and diagrams generated by explicit statements may also need to be positioned. For example, when a table is generated using the tabulation and indentation facilities, it may be desirable to prevent it being split across pages and to place it at the top or bottom of a page as with diagrams. The first need is met by @Q which is followed by an integer specifying the number of lines required by a following table. If the specified number of blank lines do not remain on the current page a new page is forced otherwise the @Q has no effect. The second more complicated requirement is met by two warning sequences @A and @Z. @A should appear in front of any sequence of text, but most often a table encoding, and it should be followed by an integer giving the amount of space in lines that the text will occupy in the output. The given text whose end should be marked with @Z will not be output until there are only just enough lines remaining on the current page. If the number of lines remaining is too small it will be output at the head of the next page.

Another way in which tables and diagrams can be generated explicitly by in-line commands is by calling another procedure from TEXT using the @* command. This may be followed by any job control command. In particular, if a flowchart is to be inserted it might be required to call the DRAW procedure described in 6.2.7.

Finally there is a facility @Y, which allows the user to alter the format of the heading at the top of every page.

6.1.5 SUMMARY OF TEXT COMMANDS

@A	displaced text header
@B	paragraph start
@C	underline one character
@D	leave space for diagram
@E	leave displaced space
@F	end of document
@I	indentation
@L	underline one line
@M	line centring
@N	indentation following line centring
@P	new page
@Q	force new page for large tables
@S	section start
@T	tabulation stops
@V	reset layout control parameters
@W	underline one word
@X	reset warning characters
@Y	alter page heading
@Z	end displaced text
@*	
@newline	
@@	

6.1.6 THE SPELL FACILITY

This procedure is simply an aid to checking the consistency of spelling in a text file such as the TEXT procedure might generate. It requires two parameters which are the file to be checked and a dictionary. All words in the text file are compared with words in the dictionary and if a match is not found the word is noted. A word in both the dictionary and the text files is any sequence of alphabetic characters between spaces and/or newlines. The list of words whose spelling is not found in the dictionary appear as the current file. Obviously it can be printed, edited or added to the dictionary as appropriate.

```
SPELL (P1, P2)

P1 is a dictionary file name
P2 is a text file name.
```

6.2 FLOCODER

Flocoder is a system for designing, documenting and generating programs using flowcharts. A file of flowchart descriptions is created, from which the charts may be drawn on any suitable output device (lineprinter, plotter or VDU, for example). The chart descriptions may of course be edited, and the charts re-drawn as necessary.

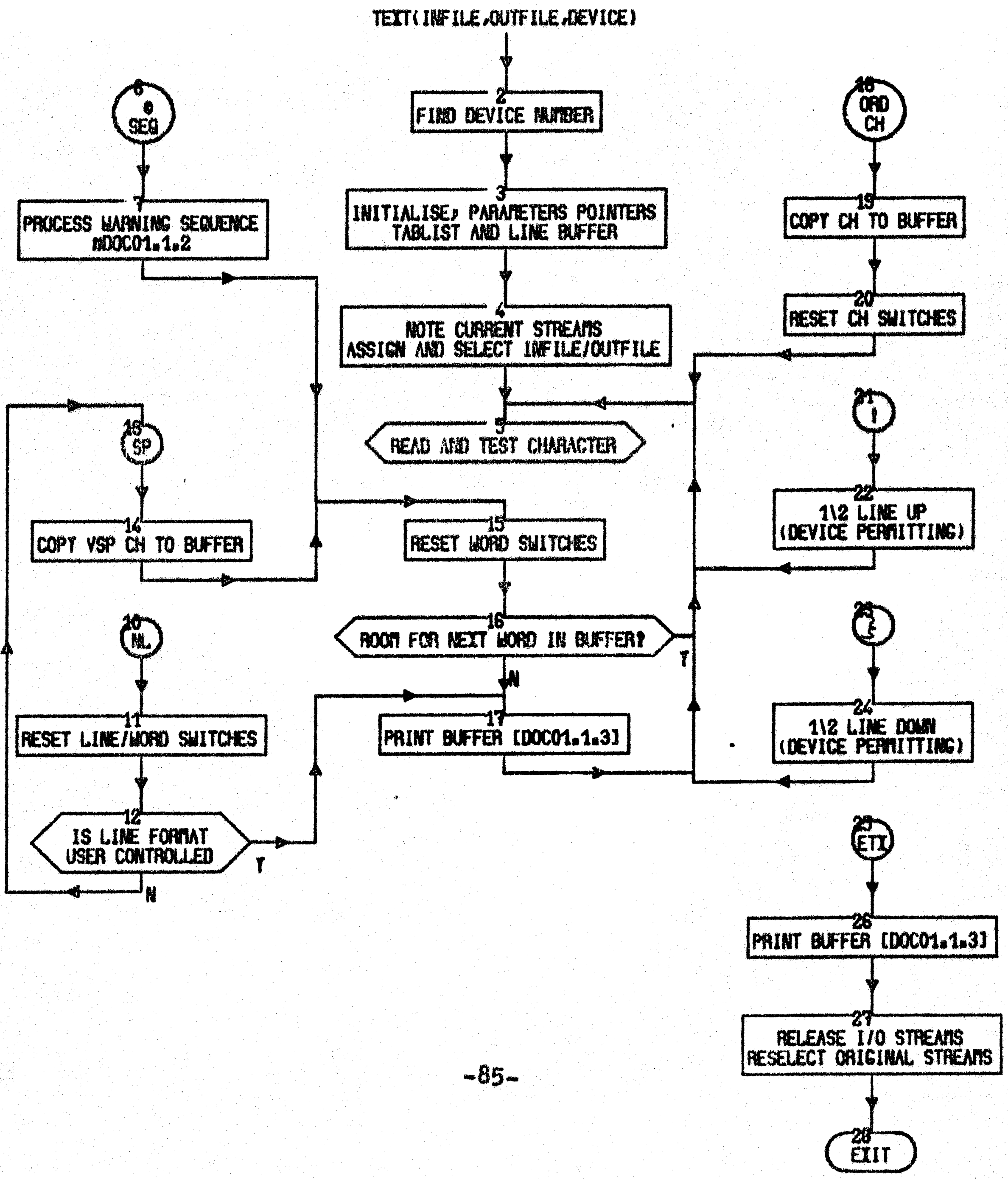
To enable Flocoder to generate or display the required program, the user provides a 'translation' for each box. If the action required in a box is simple, it will translate into a sequence of statements in a programming language; if it is complex, the translation may reference other flowcharts. In this way a hierarchy of flowcharts is created to represent the program. In fact, several translations can be given for each box. The first would normally be an English statement describing the logical function of the box and would be for display purposes only. The programming language translations, for each of the required languages, would be added later.

In effect the Flocoder system comprises a language for describing flowcharts and two procedures for processing this language. One of these 'DRAW' will draw the flowcharts. The other 'FLIP' will form a linear program by correctly ordering the boxes and adding labels and 'goto's as necessary, although this latter representation is only seen by compilers.

The syntax of the Flocoder input language is simple and straightforward. Each statement in a chart description begins with the symbol '@' as the first character of a line, followed by a keyword, and continues until the start of the next statement. The keywords can be abbreviated to single letters since they are recognised by the first letter only; after that, all characters up to the next space or decimal digit are ignored. A complete chart description consists of

A TITLE statement
One or more COLUMN statements
Zero or more ROW statements
Zero or more FLOW statements
Zero or more PARAMETER statements
One or more BOX statements
An END statement.

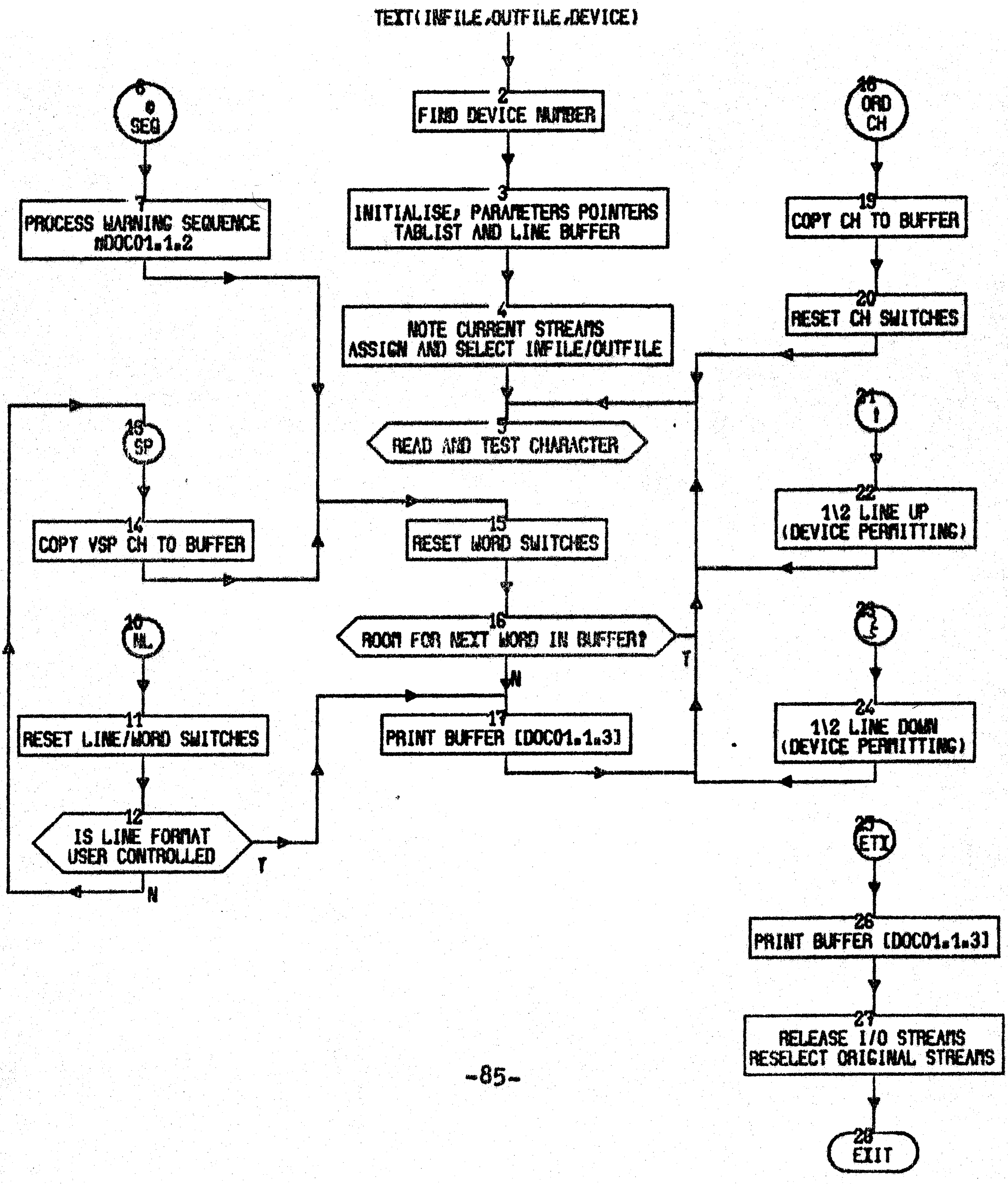
These individual statements are described below and are illustrated by examples taken from the encoding of the following diagrams.

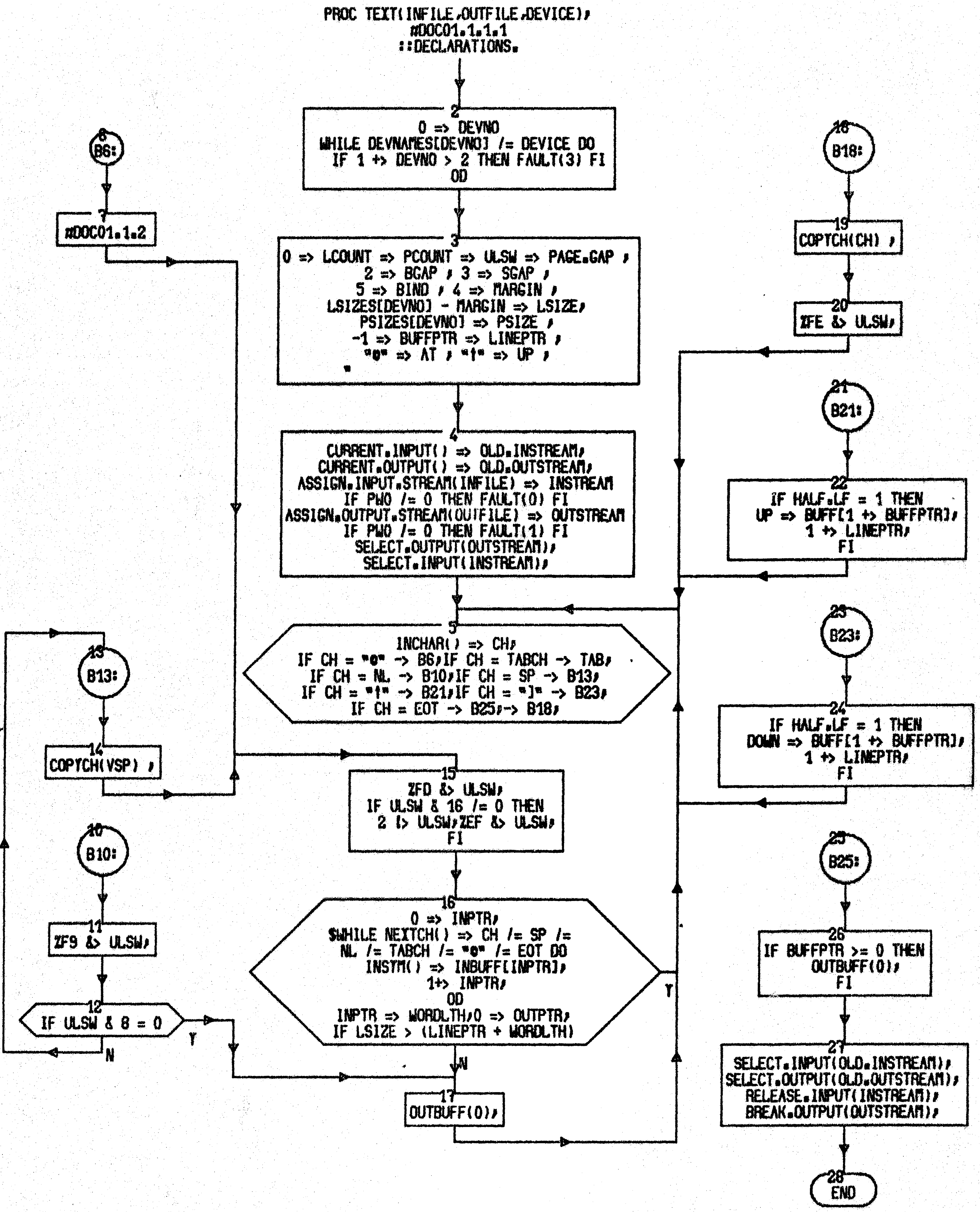

TEXT(INFILE,OUTFILE,DEVICE)
2 FIND DEVICE NUMBER
3 INITIALISE, PARAMETERS POINTERS TABLIST AND LINE BUFFER
4 NOTE CURRENT STREAMS ASSIGN AND SELECT INFILE/OUTFILE
5 READ AND TEST CHARACTER
15 RESET WORD SWITCHES
16 ROOM FOR NEXT WORD IN BUFFER?
Y
N
17 PRINT BUFFER [DOC01.1.3]
6 @ SEQ
7 PROCESS WARNING SEQUENCE mDOC01.1.2
13 SP
14 COPY VSP CH TO BUFFER
10 NL
11 RESET LINE/WORD SWITCHES
12 IS LINE FORMAT USER CONTROLLED
Y
N
18 ORD CH
19 COPY CH TO BUFFER
20 RESET CH SWITCHES
21 !
22 1\2 LINE UP (DEVICE PERMITTING)
23
24 1\2 LINE DOWN (DEVICE PERMITTING)
25 ETI
26 PRINT BUFFER [DOC01.1.3]
27 RELEASE I/O STREAMS RESELECT ORIGINAL STREAMS
28 EXIT

keyword, and continues until the start of the next statement. The keywords can be abbreviated to single letters since they are recognised by the first letter only; after that, all characters up to the next space or decimal digit are ignored. A complete chart description consists of

A TITLE statement
One or more COLUMN statements
Zero or more ROW statements
Zero or more FLOW statements
Zero or more PARAMETER statements
One or more BOX statements
An END statement.

These individual statements are described below and are illustrated by examples taken from the encoding of the following diagrams.





```
PROC TEXT(INFILE,OUTFILE,DEVICE);
#DOC01.1.1.1
::DECLARATIONS.

2
0 => DEVNO
WHILE DEVNAMES[DEVNO] /= DEVICE DO
  IF 1 +> DEVNO > 2 THEN FAULT(3) FI
OD

3
0 => LCOUNT => PCOUNT => ULSW => PAGE.GAP ,
2 => BGAP , 3 => SGAP ,
5 => BIND , 4 => MARGIN ,
LSIZES[DEVNO] - MARGIN => LSIZE,
PSIZES[DEVNO] => PSIZE ,
-1 => BUFFPTR => LINEPTR ,
"@" => AT , "↑" => UP ,
.

4
CURRENT.INPUT() => OLD.INSTREAM,
CURRENT.OUTPUT() => OLD.OUTSTREAM,
ASSIGN.INPUT.STREAM(INFILE) => INSTREAM
IF PWO /= 0 THEN FAULT(0) FI
ASSIGN.OUTPUT.STREAM(OUTFILE) => OUTSTREAM
IF PWO /= 0 THEN FAULT(1) FI
SELECT.OUTPUT(OUTSTREAM),
SELECT.INPUT(INSTREAM),

5
INCHAR() => CH,
IF CH = "@" -> B6,IF CH = TABCH -> TAB,
IF CH = NL -> B10,IF CH = SP -> B13,
IF CH = "↑" -> B21,IF CH = "]" -> B23,
IF CH = EOT -> B25,-> B18,

6
B6:

7
#DOC01.1.2

10
B10:

11
%F9 &> ULSW,

12
IF ULSW & 8 = 0

13
B13:

14
COPYCH(VSP) ,

15
%FD &> ULSW,
IF ULSW & 16 /= 0 THEN
  2 !> ULSW,%EF &> ULSW,
FI

16
0 => INPTR,
$WHILE NEXTCH() => CH /= SP /=
NL /= TABCH /= "@" /= EOT DO
  INSTM() => INBUFF[INPTR],
  1+> INPTR,
OD
INPTR => WORDLTH,0 => OUTPTR,
IF LSIZE > (LINEPTR + WORDLTH)

17
OUTBUFF(0),

18
B18:

19
COPYCH(CH) ,

20
%FE &> ULSW,

21
B21:

22
IF HALF.LF = 1 THEN
UP => BUFF[1 +> BUFFPTR],
  1 +> LINEPTR,
FI

23
B23:

24
IF HALF.LF = 1 THEN
DOWN => BUFF[1 +> BUFFPTR],
  1 +> LINEPTR,
FI

25
B25:

26
IF BUFFPTR >= 0 THEN
  OUTBUFF(0),
FI

27
SELECT.INPUT(OLD.INSTREAM),
SELECT.OUTPUT(OLD.OUTSTREAM),
RELEASE.INPUT(INSTREAM),
BREAK.OUTPUT(OUTSTREAM),

28
END
```

6.2.1 THE TITLE STATEMENT

Example:

```
@TITLE DOC01.1.1
```

The TITLE statement indicates the start of a new chart, and gives a title, which serves two functions. First, it appears on the flowchart whenever it is drawn, and thus serves to identify the drawing. Second, it is used in cross references within the code. A chart title may consist of any sequence of characters, terminated by a newline symbol. By convention, however, they are usually chosen so as to provide an index into the software. Thus the title in the example above is the first subchart of section 1 of the documentation packages.

6.2.2 THE COL STATEMENT

Example:

```
@COL 1S-2R-3R-4R-5T-15R-16T-17R
```

The column statements provide, for each box: a numeric identifier in the range 1-63, the type (shape) of the box, and the position of the box on the flowchart. A chart may contain up to eight columns. The first column statement describes the leftmost column, and the last one the rightmost column. If there is more than one box in a column, the first one specified is the highest in the column and the last one the lowest, etc. The box types, which follow the box numbers, consist of single letters with the following meanings

Letter	Meaning
A	Annotation box (no outline)
C	Circle box (used for external flows)
F	Finish box (lozenge outline)
N	Null box (a point)
R	Rectangle box
S	Start box (no outline)
T	Test box

6.2.3 THE ROW STATEMENT

Example:

```
@ROW 6-2-18
```

Each row statement gives a list of boxes to be horizontally aligned. The ordering of the box numbers in the row statements has no significance. Normally the boxes within a column are placed a minimum distance apart, and may be imagined as being connected to the box above (if any), or to the top of the diagram (in the case of the first box of a column) by invisible elastic. This means that boxes tend to be as high in their columns as possible. The effect of the ROW statement is to force horizontal alignment by 'stretching the elastic'.

6.2.4 THE FLOW STATEMENT

Example:

```
@FLOW 10-11-12N-13-14-15-16N-17-5
```

These statements specify the logical interconnections of the boxes. Text which is to appear at the point where a flowline leaves a box may also be specified in the flow statements. Any string of characters excluding newline and terminated by a hyphen is allowed. Except for test boxes, which may have two, there should be not more than one flowline leaving each box. Of course, the finish box will have no flow out.

6.2.5 THE BOX STATEMENT

Examples:

```
@BOX 2.0
FIND DEVICE NUMBER
@BOX 2.1
0 => DEVNO;
WHILE DEVNAMES[DEVNO] /= DEVICE DO
    IF 1 +> DEVNO > 2 THEN FAULT(3)
    FI
OD
```

These statements specify the text contained within each box, which may consist of any number of lines up to the start of the next statement. Several 'translation levels' maybe defined for each box, corresponding to translations in several different languages. The example above gives translations in English at level 0 and the system design langauge (MUSL) at level 1. When the charts are drawn any translation level can be selected for display in the boxes. The first chart was produced by specifying level 0 (English) and the second by specifying level 1 (MUSL). Similarly, the procedure FLIP can be instructed to generate code from any translation level.

In addition, it is possible to specify alternative translations for a box at a particular level. This facilitates the production of several versions of a program (in the same language) which differ in only a few boxes. It is a useful feature when the program is to be compiled for several machines. Alternatives are defined by appending an 'alternative number'. Thus

```
@BOX 3.2.4
```

specifies alternative 4 of level 2, box 3. The procedure FLIP may be instructed to select a particular alternative wherever it is defined, and the default (zero) elsewhere.

Flowchart cross-references may be inserted in the code by giving the name of the chart to be included, preceded by the character # at the start of a line, thus

```
#DOC01.1.2
```

appears in the translation for BOX 7 of DOC01.1.1. As a result, the code for these subcharts will be inserted in DOC01.1.1 whenever code is generated for it.

A chart reference may also, where appropriate, include parameters consisting of character strings, enclosed in brackets, and separated by commas; thus

```
#TITLE(ABC,DEF)
```

Inside the chart definition warning characters can be used to reference the parameters. Each time code is generated for such a chart the actual parameter will be substituted.

6.2.6 THE END STATEMENT

Example:

```
@END
```

This statement terminates the description of a flowchart.

6.2.7 FLOCODER PROCEDURE SPECIFICATIONS

There are three procedures in the flocoder system as follows

1) FLIP([C],I,I,[C],[C],[C])

This procedure converts the flowchart specifications on the specified input file into a linear program corresponding to a particular level of coding, as outlined in the preceding description. The next item on the input stream which is selected at the time it is called should be the title of the chart to be coded. The linear program generated is defined as the current file.

P1 - name of the input file (or stream, see Chapter 3). If this is left unspecified, the current file is used. If no current file is defined, the currently selected input stream is used.

P2 - the required coding level. If a particular alternative at this level is required, it should be encoded as: 8 bits alternative, 8 bits level - e.g. %0302 is alternative 3 of level 2.

P3 - Print switch,
if this is = 0 minimal monitor printing occurs
if it is = 1 full monitor printing occurs and if it is = 2, the code generated is listed on the currently selected output stream.

P4 - a string giving the form of labels required.

P5 - a string giving the form of 'goto's required.

P6 - a string giving the form of conditional 'goto's required.

In the strings P4, P5, P6, a unique numeric identifier is inserted immediately prior to the last character. If any of P4-P6 are zero, the following defaults are assumed:

P4 - LB: giving labels of the form LBXXXXX:

P5 - -> LB!OA! giving gotos of the form -> LBXXXXX newline

P6 - -> LB!OA! giving conditionals of the form -> LBXXXXX newline

2) DRAW([C],I,I)

This procedure produces on the current file directives to draw selected charts from the input file P1 on a graphical output device. The directives may be either sent to a graphical output device controller (e.g. PLOT), or used as input to the procedure PIC described below which generates a pictorial representation on non graphical output devices.

A list of titles of the charts to be drawn should appear on the currently selected input stream, separated by newlines and terminated by the charater ' @ ' at the start of a line. The word ALL will suffice if it is required to draw all flowcharts on the specified file.

P1 - name of the input file (or stream, see Chapter 3). If this is left unspecified, the current file is used. If no current file is defined, this procedure will expect the Flowchart encodings to be on the current stream following the list of titles.

P2 - the required level number. As with FLIP, if a particular alternative is required it should be encoded as 8 bits alternative, 8 bits level number.

P3 - specifies the device type on which the flowcharts are to be drawn as follows

```
= 0 ordinary printer (LPT)
= DBL diablo printer
= LPT line printer
= PLT pen plotter.
```

CHAPTER 7. ERROR SIGNALLING AND RECOVERY.

This chapter describes the facilities available to the programmer for signalling, detecting and acting upon error and other exception conditions. Detailed information on the error codes and error messages produced by the system is given in Appendix 2.

7.1 ERROR DETECTION AND SIGNALLING.

An error condition indicates the failure of some piece of hardware or software to carry out the task it has been called upon to do. Usually, the program requesting the operation will require to be informed of this failure, in order either to effect a local recovery action or to inform its caller in turn that an error has occurred. Thus a general mechanism is needed whereby a called program (or a hardware operation) may inform its caller of the existence and nature of any errors which may arise.

There are two possible approaches to the problem of signalling detected errors to a calling program. One is to set an error code into a status variable, which may subsequently be inspected by the caller, and then to return to the caller. Alternatively the caller may be "trapped" - i.e. forced into an error handling procedure which it has previously nominated to deal with the condition. Obviously, each method has its advantages: the former is often easier to use, particularly for errors which are expected to occur (e.g. a user mis-typing a parameter); on the other hand, the trapping method is potentially more efficient since it eliminates the need to check the status after each action which might cause an error.

MUSS provides for both of these methods, and allows the programmer to select which is to be used for particular classes of error.

7.2 CLASSIFICATION OF ERRORS.

For convenience of recovery, errors are grouped into a number of error classes, and the recovery action may be specified separately for each error class. Within the classes, individual errors are identified by an error code; a complete list is given in Appendix 2.

The basic system distinguishes sixteen different error classes numbered 0 - 15. The first ten of these are system defined; classes 10-15 are available for use by applications software. The ten system-defined error classes are:

0 - program fault
1 - limit violation
2 - timer runout
3 - external interrupt
4 - organisational command error
5 - Input/Output setup errors
6 - Input errors
7 - Job control command errors
8 - Programming language runtime errors
9 - Basic applications errors.

7.2-0 Program Faults.

This class consists mainly of errors detected by hardware during instruction execution - for example, arithmetic overflow, access to illegal store addresses, etc.

7.2-1 Resource Limit Violations.

These errors are signalled by the system kernel when a process exceeds one of its stated resource limits - for example, processing time.

7.2-2 Timer Interrupts.

These are not strictly errors at all, but are handled by the same trapping mechanism. The process may request to be informed after it has used a specified amount of processing time (see System Programmers' Manual). The mechanism is used by the basic system to obtain an "early warning" of the processing time limit violation, by requesting a timer interrupt shortly before the actual time limit is exceeded.

7.2-3 External Interrupts.

These are not strictly errors at all, but are handled by the same mechanism. An external interrupt is triggered by some other process (see System Programmers' Manual), usually a device controller, and its main use is to implement the "break-in" facility for interactive jobs.

7.2-4 Organisational Command Errors.

The organisational command procedures which form the interface with the MUSS kernel all signal errors via this error class.

7.2-5 Input/Output setup error.

Errors in the organisation of input/output streams are included in this error class.

7.2-6 Input errors.

The errors in this class are those which commonly result from faulty input, rather than a faulty program - for example, reading beyond the end of a file, or reading non-numeric data in a context where numeric data is expected.

7.2-7 Job control command errors.

This class is reserved for errors detected by the job control command interpreter.

7.2-8 Programming language runtime errors.

This class consists of language dependent errors, such as accessing outside the bounds of an array or calling an undefined procedure. To the user they are logically an extension of class 0.

7.2-9 Basic Application Errors.

All errors detected by the applications packages of the basic system (i.e. those described in this manual, such as EDIT, TEXT, etc.) fall into this class.

7.3 PROCEDURES FOR ERROR HANDLING.

1) SET.TRAP(I,P)

This procedure nominates P2, which should be a procedure with two parameters, as the trap procedure to be involved on detecting an error in class P1.

2) READ.TRAP(I)P

This procedure returns as its result a reference to the current trap procedure for errors of class P1.

3) SET.RECOVERY.STATUS(I,I)

This procedure is used to control whether the "trapping" or "status" mechanism is to be used for errors of class P1. P2 specifies the mechanism to be used: zero means trapping, 1 means status.

4) READ.RECOVERY.STATUS(I)I

This procedure returns the current recovery status for error class P1.

5) ENTER.TRAP(I,I)

This procedure signals the occurrence of the error with class P1, code P2. Depending upon the recovery status in force for error class P1, the error will be signalled either via a trap or via the global status parameter PWO.

7.4 PROCEDURES FOR FAULT MONITORING.

1) OUT.M(I,I)

This procedure output, on file the currently selected stream, a suitable error message for error class P1 code P2.

2) OUT.T(I,I)

This procedure outputs, on the currently selected stream, a standard trap diagnostic for error class P1 code P2 (i.e. an error message and other diagnostic information).

7.5 NOTES ON THE HANDLING OF ERRORS.

1) Error classes 0-3 are implemented within the operating system kernel, and are always handled via the trapping mechanism (SET.RECOVERY.STATUS) should not be used).

2) Error class 4 may be handled by either trapping or status, but most of the system library procedures assume that the status mechanism will be used. Thus some library operations may not work correctly if trapping is activated for error class 4.

3) Library procedures which alter the trap procedures or the recovery status should restore their original values prior to exit. Errors which have been detected may be "passed back" to the caller by restoring the original trap procedure and status values, and then re-calling ENTER.TRAP.

4) If the "status" form of error recovery is active, then an appropriate error code is set into the global status variable PWO, and the ENTER.TRAP procedure returns immediately. Thus, in cases where status recovery may be in use, calls of ENTER.TRAP should always be followed immediately by a return to the calling program which may then inspect the status variable.

Generally, use of the status form of recovery is only recommended in small localised code sequences.

5) The status variable PWO is encoded as two bytes, the most significant giving an error class, and the least significant an error code.

7.6 HIGH LEVEL LANGUAGE RUN TIME DIAGNOSTICS.

At the start of running a high-level language program the traps are set to enter the high-level language dump and on-line debugging facility (MURD). This is a language independent package for use with the standard MUSS language, MUSL, Pascal and Fortran 77. The procedure:

RD.TRAP(I,I)

is entered initially with error class P1 and code P2. This prints the reason for the program failure, the line number of the line where the failure occurred and the procedure in which it occurred.

If the program is running off-line the procedure

```
RD.DUMP()
```

is then entered.

This prints the variables of the current procedure. It then prints the line number and procedure where the current procedure was called and the variables of that procedure. This is repeated until the outermost level of the program is reached.

If the program is running on-line the procedure

```
RD.DEBUG()
```

is then entered.

This procedure provides a simple user interface to the MURD run time diagnostic facilities. The detailed specification of these facilities, given in the MUTL manual, is intended for users wishing to implement their own version of RD.DEBUG.

The RD.DEBUG user interface allows users to type

(a) run-time diagnostic commands.
(b) names of variables to be inspected or changed.
(c) MUSS commands.

RUN TIME DIAGNOSTIC COMMANDS.

These commands are identified by a single letter.

A All : prints names and values of all variables of the current procedure.

N Next : move to next procedure out in the dynamic chain.

C Current : move back to the current procedure.

D Dump : the RD.DUMP procedure to give a complete traceback of variable values.

G Go : restart the program.

Q Quit : end run-time diagnostics and revert to job control command processing.

V Variable : move into variable inspecting and changing mode.

** Treat the line as an MUSS job control command and execute that command.

VARIABLE INSPECTION AND CHANGING.

<variable name><space> causes the named variable to be printed.

<variable name>/<value> allows a new value of the variable to be entered from the terminal.

After a value has been printed a new value of the same variable may be entered.

* command letter reverts to diagnostic command mode.

** causes the line to be treated as an MUSS job control command and that command is executed.

MUSS COMMANDS.

As indicated above any MUSS command on a line by itself preceded by ** is interpreted and executed immediately.

CHAPTER 8. HIGH LEVEL LANGUAGES, LIBRARIES AND COMPILED PROGRAMS.

8.1 HIGH LEVEL LANGUAGES

Compilers for the following languages are available under MUSS :

FORTRAN 77
PASCAL
COBOL
MUSL

Other languages may also be implemented.

The normal method of running a high level language program is to call the compiler and, if the compilation is correct to call the RUN procedure to execute the program. Compilers for the languages listed above exist as procedures in the system library. The five parameters of compiler procedures are:

P1 = Document on which source text is found.
P2 = Document to which compiler output is sent.
P3 = Mode, zero normally. See 2.5.2 for its use and under particular compiler for special modes.
P4 = Library information, see 2.5.2 for details.

The procedures are:

FORTRAN([C],[C],I,I)

The language implemented is Standard Fortran 77. See Chapter 10 for a full description.

PASCAL([C],[C],I,I)

The language implemented is Standard Pascal. See Chapter 12 for a full description.

COBOL([C],[C],I,I)

See Chapter 11.

MUSL([C],[C],I,I)

The basic system library is automatically opened for every process on creation. The LIBRARY command allows further libraries to be opened. Public libraries are opened by giving the library name as parameter of the LIBRARY command, the user does not need to know whether the library is preloaded into the MUSS virtual store. For private libraries the library name must be a filename in a currently open file directory.

The directory of the most recently opened library is searched first by the FINDN procedure. This enables selective replacement of procedures in existing libraries.

A public or private library is either opened in reference or execute mode. In reference mode only the directory structure is opened so that references to its procedures can be verified during compilations. In execute mode the library code is opened, and any initialisation code of the library executed, and any global space of the procedure allocated.

```
Bit 0 = 0  Open the library in execute mode
Bit 0 = 1  Open the library in reference mode.
```

LIBRARY([C],I)   LIB

P1 The name of the public library or file containing the private library to be opened.

P2 MODE

```
Bit 1 = 0  Open library directory and make procedures available.
           This is the normal mode of library open.
Bit 1 = 1  Open library if not already open. Normally a new
           (possibly modified) copy of a library is
           obtained by the LIBRARY command. With P2=2 a
           library already opened is not re-opened.
```

DELETE.LIB([C])   DL

This closes the library referenced by P1 and unlinks its directory from the directories currently open.

If P1=0 all libraries except the basic system library are closed.

DELETE.PROG()

Deletes the current program.

LIB.LOAD([C])

The purpose of this procedure is to load compiled library procedures into a public library so that they may be available to all users (whereas LIBRARY only makes procedures available to the current process).

P1 The file created by a compilation containing the code and directory of the library.

CHAPTER 9.1 THE OVERALL ORGANISATION OF MUSL

9.1.1 INTRODUCTION

The overall structure of MUSL reflects its principle function, namely the implementation of MUSS. In designing such a language for operating system implementation one of two approaches can be taken. First an attempt can be made to formally cater for the operating system requirements in the 'virtual machine' provided by the langauge. For example, the notion of tasks (or processes) and multi-tasking can be included. However, this tends to move some of the operating system functions into the runtime package of the language rather than providing the means for their implementation. The second approach, which is the one followed in MUSL, is to allow direct access, through the language, to the actual machine. Thus an operating system written in MUSL controls directly the use of store, CPU and peripherals. Many of the special features of MUSL will not be required for writing simple user programs, and the underlying basic language is straightforward and Pascal-like.

For the purpose of describing the syntax of MUSL this manual uses a variant of BNF. This will be mostly self evident to readers familiar with this kind of notation. For example

```
<A> ::= W!X[Y!Z]
```

means that the syntactic element A may be W, XY or XZ. Also

```
<B> ::= <A>[<B>!<NIL>]
```

means that B may be

```
   W or XY or XZ
or WW or WXY or WXZ
or XYXY or XYW or XYXZ
or XZXZ or XZW or XZXY
or WWW or WWXY or WWXZ
      and so on.
```

NIL is the only reserved name used, but some notation is required for the symbols <,>, !, [,] and this is <<>,<>>, <!>, <[>, <]>.

9.1.2 MODULAR STRUCTURE

Software written in MUSL is normally subdivided into modules. A single module is all that the compiler will accept. In the simplest case a module can be a complete program, but more commonly a program will consist of several modules and MUSS consists of many modules. Since the MUSL compiler only deals with single modules, it follows that the compilation of multi-module software will involve several calls on the MUSL compiler (one for each module), and that provision for inter-module linking has to be made outside of the compiler. This Manual is mainly concerned with the rules for writing single modules but some discussion of module linking is appropriate to set the context.

9.1.3 MODULE LINKING

Like all other compilers in the MUSS system the MUSL compiler generates code indirectly through calls on the MUTL (target language) procedures. These procedures can be instructed to generate either relocatable or executable binary. In the former case inter-module linking is postponed until the final link-loading pass, whereas in the latter case it becomes the responsibility of the MUTL procedures, and is carried out as the compilation proceeds. The MUTL procedures and the link-loading scheme are described elsewhere but to give an understanding of how modules of MUSL fit together some repetition is appropriate here. It will be sufficient to consider the case where the MUTL procedures are generating executable binary, and hence performing the inter-module linking.

9.1.4 THE STORAGE MODEL

Statements described under 9.4.1 allow a user to separate both the code and the data, contained in a module, into <u>areas</u>. This facility might be used in a module of the operating system, for example, to separate the code that needs to be resident in main store from that which might be paged, and similarly for the data structures. The MUTL system provides the means for areas to be placed in physical store.

MUTL considers the physical store to be subdivided into <u>segments</u>. Each of these has a number, for identification purposes, a size, a runtime address and a compile time address (all in units of bytes). They are defined by calling the MUTL procedure TL.SEG. In fact TL.SEG has two further parameters as described in the MUTL Manual, but their function is not appropriate to understanding MUSL. Identification numbers are allocated consecutively from 0 in the order in which the TL.SEG calls occur. Areas, are related with segments by the MUTL procedure

TL.LOAD (MUTL segment number, MUSL area number).

Area 0 has a special significance of which the user must be aware. It is the runtime procedure stack (9.2.5), whose placement is determined by the MUTL system rather than by the user.

After the storage requirements have been specified for a set of modules that require to be linked, they are compiled by a sequence of calls on the MUSL compiler one for each module. Unless the defaults described in 9.1.7 are adequate each module should be preceded by calls on TL.LOAD to cause proper placement of its Areas. As the compilation proceeds, the code and data items of the areas will accumulate in their respective segments. If more than one area of a module is mapped into a given segment, some interleaving may occur, depending upon the placement of area selection directives (9.4.1) in the source.

9.1.5 INITIALISATION

It will be obvious from what has gone before that the MUTL procedures will require data structures that carry information across many individual MUTL procedure calls. These data structures must be initialised, therefore at the start of a compilation the procedure TL (MODE) should be called. Also there are some final checks required after the last module of a compile job and the procedure TLEND must be called.

9.1.6 MODES OF COMPILATION

For a 'normal' compilation of a user program the value 0 for the MODE parameter of TL selects appropriate options. The full range of options are described in the MUTL Manual.

9.1.7 THE COMMAND STRUCTURE OF A TYPICAL COMPILE JOB

In the most general case the sequence of commands required to control a compilation would be

a call on TL
one call on TL.SEG for each segment
one call on TL.LOAD for each area of the 1st module
a call on MUSL to compile the 1st module
a further sequence of calls on TL.LOAD followed
by a call on MUSL for each additional module
finally a call on TL.END.

When TL is called with a 'normal' MODE, segment 0 is automatically created (for code) and area 1 is automatically mapped into it at the start of each module. As stated earlier, area 0 is the procedure stack, and its placement in store is controlled by MUTL. That is area 0 cannot be mapped into a user created segment. Thus, for a simple program TL.SEG and TL.LOAD calls are not necessary.

Also unless directives (9.4.1) are used to control the placement of code and variables, all code will be placed in area 1 (segment 0) and variables will be placed in area 0 (the stack). In more complicated situations segments are created at the start of a compilation, and the mapping of areas into segments is given separately for each module.

9.1.8 THE FORMAT OF A MODULE

In the general case a module will use procedures, and possibly literals, data structures, types and labels declared in other modules. Declarations for these entities, known as the 'imports' to the module, must precede the module heading. The module heading itself serves three functions. First it indicates the start of the module and the entities declared after the heading are private to the module unless they are formally 'exported'. The second function is to allow these exports to be specified, as a list of names enclosed in parentheses. The third function is to provide the option of assigning a name to the module. If this option is used then all references to the exports from the module in other modules must prefix the name of the exported entity by the module name. A module is terminated by '*END' and on encountering this the compiler will 'exit'.

More formally the syntax of a module is

```
<MODULE> ::= <IMPORTS>
             MODULE[<NAME>!<NIL>]<EXPORTS>
             <STATEMENTS>
             *END
```

The IMPORTS are declarations of entities exported from other modules. Their exact form will be discussed in 9.6, however, it should be noted that they are of two kinds. If the full detail of an imported entity is needed by the compiler, in order to compile references to it inside the module, a full duplicate declaration must be used (as in the above example). If there is no relevent detail a special import declaration (9.5.10) is used that gives only the name and kind of the entity.

The EXPORTS of a module are an optional list of names of entities declared within the module thus

```
   <EXPORTS> ::= (<NAME.LIST>)!<NIL>
<NAME.LIST> ::= <NAME>[,<NAME.LIST>!<NIL>]
```

For example, a module M2 that imports a procedure P1, which is exported from another module M1, and exports itself two procedures P2, P3 would have the general form

```
PSPEC M1.P1(INTEGER)
MODULE M2(P2,P3)
```

```
PSPEC P2(INTEGER)
PSPEC P3(INTEGER)
PROC P2(I)
   .
   .
statements of P2
   .
   .
END
PROC P3(J)
   .
   .
statements of P3
   .
   .
END
*END
```

It is assumed in the example that P1, P2, P3 each have one integer parameter. Obviously any external references to P2 and P3 must use the names M2.P2, M2.P3.

STATEMENTS are discussed more fully in subsequent Sections but it might be helpful to give an approximate picture of how they relate to modules and programs at this point. They fall into two main groups, declaratives and imperatives. The norm is for the declaratives to come first. Again the norm will be for most of the statements in a module to be concerned with defining procedures. That is, they will be contained within procedure bodies that are delimited by the procedure headings and the matching ENDs. Thus a module will typically take the form

```
import declarations
module heading
declaratives
imperatives
     procedure heading
     procedure body
     END
       .
       .
       .
*END
```

9.1.9 KINDS OF MODULES

There are now several cases to consider in order to convey the significance of modules.

If the module is an ordinary program, a run of the program implies executing the imperative statements, in the main body of the module. These may call the procedures enclosed in the module which will cause the imperatives that they contain to be executed. Thus the imperatives at the outer level might be regarded as the 'main program'. A simple variant of this case is where the module is one of several that make up a complete program. In this case the imperative parts of each module will be joined together in the order in which they are loaded. In most of the modules the imperative statements, in their main body, will usually be concerned with initialising the global data structures of the module, whilst that of one module will form the main program. Clearly the user must be conscious of the order of execution.

Sometimes in a program that consists of several modules, some of the modules may be entirely passive, in that they contain no imperatives in their main body. In this case the only way that the procedures of the module can be executed is through calls of the exported ones, occuring in other 'active' modules.

Modules of this kind can be treated as libraries, which are compiled and filed, to be repeatedly used by other programs after being 'opened' by use of the LIB command. The main body of a library module may in fact contain imperatives statements, in which case they will be executed when the library is opened. Their function, therefore, is to initialise the data structures of the module.

Some modules form part of the MUSS system library. They are permanently 'open', hence they cannot have initialisation statements. If initialisation is required, as in the case of the MUTL procedures, a procedure (e.g. TL) must be provided that can be explicitly called by the user.

9.2 KINDS OF STATEMENTS AND SCOPE RULES

9.2.1 KINDS OF STATEMENT

There are a number of places in the language specification where an arbitrary sequence of statements occur. Thus there is a need for the definition

```
          <STATEMENTS> ::= <STATEMENT>[<STATEMENTS>!<NIL>]
```

This section is concerned with various kinds of STATEMENT that exist in the language. It will be apparent that not all kinds of statement are appropriate in every context and norms will be suggested. However, the detailed constraints will only be given as the individual statements are described in later chapters.

There are five main kinds of STATEMENT as follows

```
          <STATEMENT> :: = <DECLARATIVE.STATEMENT>!
                           <IMPERATIVE.STATEMENT>!
                           <DIRECTIVE.STATEMENT>!
                           <PROC.DEFN>!
                           <BLOCK>!
```

9.2.2 DECLARATIVES

The declarative statements normally appear at the start of a MODULE, PROCEDURE or BLOCK. With the single exception of LABELS (9.5) a declaration must be given for every name introduced into a module, before any other use of the name is permitted. The main function of declarative statements is to create entries on the name and property lists of the compiler. They do not, directly, result in the generation of executable code although they may contribute to the code, executed at procedure entry and exit time, that is concerned with dynamic storage allocation.

9.2.3 IMPERATIVES

These statements are the means by which the computations are specified. Normally each MODULE, PROCEDURE and BLOCK will have a sequence of imperative statements following its declarative statements. They are fully discussed in 9.7, but briefly they consist of the usual forms of statement such as:

statements that express computations

procedure calls

FOR/WHILE loops

IF/THEN/ELSE statements

switches

9.2.4 DIRECTIVES

These statements relate more directly to the compiler and the underlying MUTL than to the language. Their function is to select 'modes' of compilation. Some directives merely set switches in the compiler, whilst others allow the environment of the compilation to be manipulated by means of direct calls on the MUTL procedures and other MUSS Library Procedures. Obviously the positions in which they are placed must be carefully chosen in relation to the action they have. Some guidance on this is given along with their descriptions in 9.4.

9.2.5 PROCEDURE DEFINITIONS

A procedure has to be both declared and defined. Procedure declarations, which are described in 9.5, specify the name of a procedure, and the number and type of its parameters and result. A procedure definition gives the code body of the procedure that is to be executed on each call.

A procedure is defined by a heading followed by a sequence of statements terminated by END.

```
<PROC.DEFN> ::= <PROC.HEADING>
                <STATEMENTS>
                END
```

Normally the STATEMENTS will consist of DECLARATIVES followed by IMPERATIVES.

The function of the PROC.HEADING is to give the name of the procedure and the names of any parameters that it may have.

```
<PROC.HEADING> ::= PROC<NAME>[(<NAME.LIST>)!<NIL>]
```

At runtime each call of a procedure causes space to be allocated dynamically on a 'procedure stack' containing linkage information, parameters, and the variables declared within the procedure. Hence procedures may be recursive.

Although every procedure must be declared before it can be used or defined, the definition need not precede calls on the procedure. The only restrictions on the placement of the definition of a procedure are that it must come after the declaration of the procedure and be at the same level of the block structure (see 9.2.7).

9.2.6 BLOCKS

A BLOCK is a sequence of statements delimited by BEGIN and END.

```
<BLOCK> ::= BEGIN
            <STATEMENTS>
            END
```

Normally the statements will consist of IMPERATIVES possibly preceded by some DECLARATIVES. The main purpose of the BLOCK construct is to represent a group of statements as a single statement. This is useful in some of the control constructs described in 9.7 where only a single STATEMENT is admissable. Another use of the BLOCK construct is to localise the 'scope' of the entities declared in the block to the statements that it contains.

9.2.7 SCOPE AND BLOCK STRUCTURE

The language allows blocks and procedures to be nested inside each other to any depth and the implications of this must be understood. Consider the module shown schematically below

```
MODULE
declaration of procedure A
declaration of procedure B
  .
  .
    PROC A
    declaration of procedure C
      .
      .
        PROC C
          .
          .
        END
    END
    PROC B
      .
      .
    END
*END
```

Subject to certain restrictions the overall scope rule is that the statements within a procedure may use entities previously declared in the same procedure, any enclosing procedure or the enclosing module. Thus the statements of procedures A and B may access any entity previously declared in A or B respectively or previously declared at the start of the module M. The statements of C may access any entity previously declared in C, A or the module M. In particular the statements of A may use C, A and B but the statements of B may not use C.

From the point of view of the general scope rule blocks and procedures need not be distinguished. Obviously since a block has no name and is not formally declared a different example to the above is necessary in order to illustrate the rule

```
MODULE M1
declaration of procedure A
declaration of X
     .
     .
     BEGIN
     declaration of Y
        .
        .
     statement S1
        .
        .
     END
     PROC A
     declaration of Z
          BEGIN
           .
           .
          statement S2
           .
           .
          END
     END
*END
```

In this case statement S1 may use Y, A and X but not Z. Statement S2 may use Z, A and X but not Y.

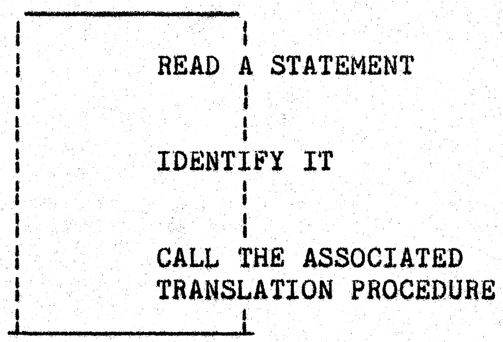
There is one important difference between the treatment of procedures and blocks that is relevant to understanding the exception to the general scope rule stated below. This difference is that on entry to a procedure, a 'frame' is created on the procedure stack that contains all the variables declared in the procedure, together with those declared in the blocks that it contains (and the blocks that they contain and so on), whereas it follows that on entry to a block there is no such action. In effect it is as if the entitie declared in a block were actually declared in the enclosing procedur (or module) head, except that access to them is restricted to th statements of the block. This interpretation should be assumed i understanding the restriction stated below regarding the scope o variable.

When nested procedures occur in a module, the only variable available to the statements of a procedure are those previously declared in either the module heading, the outermost procedure of the nest and the procedure in question.

Further restrictions apply to labels. Normally only the labels declared within a block or procedure may be used in the statements of the procedure. A special mechanism exists, as described in 9.6, that provides an escape from this restriction.

9.2.8 'STATEMENTS' AS THE COMPILER SEES THEM

From a language definition point of view the recursive approach followed above seems best. For example, a procedure definition is regarded as a statement, even though it contains an arbitrary sequence of statements some of which may be further procedure definitions. In the current implementation of MUSL, the compiler takes a different view of statements. The compiler has a main loop, which proceeds along the lines



READ A STATEMENT

IDENTIFY IT

CALL THE ASSOCIATED
TRANSLATION PROCEDURE

Obvious problems arise, on small computers such as the PDP11, if STATEMENTs are very large. Thus the compiler is organised to break certain potentially long statements into a sequence of statements. For example, a procedure statement is treated by the compiler as the group of statements

procedure heading statement
the statements of the procedure
an END statement

Clearly context information must be remembered across such a group of statements, so that when, for example, the END statement occurs the compiler knows of what it is the end. Even so this feature of the compiler influences the style of fault monitoring, described in Chapter 8, and it is responsible for some of the restrictions mentioned, in subsequent Sections, in connection with specific statement descriptions concerning the used of statement separators and newlines.

9.3. BASIC ELEMENTS OF THE LANGUAGE

9.3.1. SYMBOLS

Each statement in MUSL is a sequence of symbols. These symbols may be names, numbers or delimiters. To compensate for the shortage of suitable characters some delimiters are represented by reserved words. Alternatively an abbreviated form of each reserved word may be used comprising the first two characters of the full name preceded by the $ sign. The space character has no meaning at the statement level, but it terminates a symbol therefore it should be used to separate symbols where ambiguity would otherwise arise. Newline and tab are equivalent to spaces, except where specific restrictions are placed on the use of newlines in certain 'long statements'. The list below gives the full form of each reserved word delimiter.

IF, FI, THEN, ELSE, DO, OD, WHILE,
FOR, EXIT, SWITCH, ALTERNATIVE, FROM,
INTEGER, LOGICAL, REAL, SELECT, PROC,
PSPEC, LSPEC, LITERAL, DATAVEC, END,
BEGIN, MODULE, ADDR, SPACE, VSTORE,
LABEL, AND, OR, OF, IS, TYPE, IMPORT.

The delimiters INTEGER, LOGICAL, REAL are special in that they can be followed by an integer, giving their size in bits. Thus in effect INTEGER16($IN16) LOGICAL8($LO8) and REAL64($RE64), for example, are also delimiters.

9.3.2. STATEMENT SEPARATORS

A ';' is normally used to separate statements, it may be omitted if the statement ends with a reserved word or the following statement begins with a reserved word which cannot occur within a statement. The delimiters that start statements and can also be used within statements are

INTEGER REAL LOGICAL ADDR

Redundant ';'s between statements will be ignored, therefore if in doubt, use ; consistently as a terminator.

9.3.3. COMMENTS

The comment symbol is '::'. All following characters will be ignored up to the next newline symbol. A comment is an alternative way of terminating a statement and is equivalent to ';'.

9.3.4. NAMES

Names (henceforth referred to as <NAME>) are used to represent variables, types, literals, procedures etc. They must begin with a letter which is then followed by an arbitrary sequence of letters and digits possibly connected by fullstops (period). The latter are allowed as a readability aid and are ignored when names are matched. For example, A X X1 NAME IN.NAME1 and INNAME1 are all examples of valid names, the last two referring to the same item.

9.3.5. CONSTANTS

These (henceforth referred to as <CONST>) can be written in various styles. However, there are some "TYPE" implications, and the TYPE (and size) of the constant should match the context in which it is used, or it will be converted if the context allows type conversion (9.7.2).

```
<CONST> ::= <DEC.INTEGER>!
            <HEX.CONST>!
            <CHAR.CONST>!
            <MULTI.CHAR.CONST>!
            <CHAR.STRING>!
            <REAL.CONST>!
            <REAL.HEX>!
            <NAME>
```

9.3.5.1 Decimal Integer Constants

A decimal integer constant may be optionally preceded by a sign.

```
<DEC.INTEGER> ::= [+!-!<NIL>]<DECIMALS>
<DECIMALS>    ::= <DECIMAL.DIGIT>[<DECIMALS>!<NIL>]
```

For example, all the following are allowed

34, -90, +34, 65535

All the above constants would be appropriate in an INTEGER16 context. They could also be used in an INTEGER32 context, in which case they would be sign extended to 32 bits.

9.3.5.2 Hexadecimal Constants

Hexadecimal constants are used when the bit pattern representing a constant is more significant than its value, if interpretted as a number.

They are represented by the '%' symbol followed by sequence of hexadecimal digits, and the notation allows multiple occurances of the same digit to be specified by a repetition factor expressed as an unsigned decimal integer enclosed in brackets.

```
<HEX.CONST> ::= %<HEX.SEQUENCE>
<HEX.SEQUENCE> ::= <HEX.DIGITS>[<HEX.SEQUENCE>!<NIL>]
<HEX.DIGITS> ::= <HEXDIGIT>[(DECIMALS)!<NIL>]
<HEX.DIGIT> ::= <DECIMAL.DIGIT>!A!B!C!D!E!F
```

Hexadecimal constants are right-justified, for example, the binary patterns

1001100110010000 and 11111110

might be expressed as %9(3)0 and %FE respectively. If a hexadecimal constant occurs in context requiring more than the specified number of bits, it is automatically extended by zeros at the 'left hand end'.

9.3.5.3 Character Constants

These are used to represent the ISO-code of the visible characters. The notation is

```
<CHAR.CONST> ::= '<CHARACTER>
```

where CHARACTER represents, any printing character except $, a space character, or the character pairs

```
$L
$P
$N
$"
$$
```

representing newline, newpage, null, quotes(") or dollar ($). For example 'A, '1, '$$ are equivalent to %41, %31, %24.

9.3.5.4 Multi-character Constants

These are used when it is required to have the ISO-codes for several characters concatenated into one constant. The notation is

```
<MULTI.CHAR.CONST> ::= "<CHARACTERS>"
<CHARACTERS> ::= <CHARACTER>[<CHARACTERS>!<NIL>]
```

Examples are "NAME1", "$$RE" which are equivalent to the hexadecimal constants %4E414D4531 and %245245. Like hexadecimal constants they are right-justified and zero extended on the left when used in a context bigger than themselves.

9.3.5.5 Character Strings

When general character strings are to be passed as parameters, they should be stored in a vector of bytes and a 'reference' or pointer to the vector should be used as the parameter. Procedures that have parameters of this kind will have them declared as ADDR[LOGICAL8] (see Section 9.5.2). Since this situation occurs frequently in programs that require to print messages, a special notation is provided that allows the actual character string to be written at the point where a reference to it is required. This notation is

```
<CHAR.STRING> ::= %"<CHARACTERS>"
```

An example of its use would be

```
CAPTION (%"THIS IS AN ERROR MESSAGE");
```

9.3.5.6 Real Constant

Real constants are used to represent floating point numbers. The number of bytes occupied by the floating point number is determined by its TYPE. Floating point numbers are internally represented as a 32 or 64 bit entities.

When real constants are input, the digits up to the decimal point are evaluated as an integer constant and then converted to real. The precision of the conversion is the precision allowed for the (default) integer constant (32 bit integer). The remainder of the constant is evaluated as real. If greater precision is required, the exponent form (+1.23456789@8) must be used.

```
<REAL.CONST> ::= [<NIL>!+!-][<DECIMALS>!<NIL>].
                 [<DECIMALS>!<NIL>][@<DEC.INTEGER>!<NIL>]
```

For example

```
+23, 17.3 -90.23 -3.@9 4.3@-4 -.6666 +.5@+3
```

9.3.5.7 Real Hex Constant

REAL.HEX constants are inevitably machine dependent and should only be used if the exact bit pattern is vital to the computation. The hexadecimal value must follow the %R with no spaces.

```
<REAL.HEX> ::= %R<HEX.SEQUENCE>

                 PC
```

The internal representation of floating point numbers will be defined later.

9.3.5.8 Name Constant

Any name can be defined as a literal and given a constant value (9.4). Its use elsewhere is equivalent to using the value explicitly.

9.4. DIRECTIVE STATEMENTS

All DIRECTIVEs start with "*" which is usually followed by the name of the directive, some parameters if appropriate, and they are terminated by ;. They are used mainly to control the mapping of a module or program and its variables into areas, to control the form of compile time output and to manipulate the environment of the compilation by making calls on MUTL procedures or other library procedures.

```
<DIRECTIVE.STATEMENT> ::= *CODE<CONST>!
                          *GLOBAL<CONST>!
                          *CMAP<CONST>!
                          *STOPC!
                          *INFORM<CONST>!
                          *INIT<CONST>
                          *TLSEG<CONST><CONST><CONST>!
                          *TLLOAD<CONST><CONST>!
                          *TLMODE<CONST><CONST>!
                          *LIB[<NAME>!<NIL>]!
                          *VTYPE<TYPE>
                          **<ANY MUSS COMMAND
```

9.4.1. AREAS

Areas are the major logical subdivisions of the store in which a module is to be placed. A module can reference up to 32 areas, which at compile time are known only by number (0-31). They are assigned to actual store locations by use of the MUTL procedures TL.SEG and TL.LOAD. Area 0 is special in that it is the procedure stack, whose placement in store is directly under the control of MUTL. Code is always compiled into the area last specified in the directive

*CODE <CONST>

If the same area is specified more than once, the successive sequences of code will occur consecutively in the given area.

A precisely similar effect can be achieved with the mapping of global variables (i.e. those declared at the outer level of a module) into segments by use of the directive

*GLOBAL <CONST>

Area 0 may be specified, in which case, global variables subsequently declared will be on the stack immediately before the first stack frame. Variables declared inside procedures are dynamically allocated on the stack.

Both in the above and in later directives, the form of CONST used should be integer or hexadecimal.

9.4.2. COMPILER OUTPUT

Another pair of directives are concerned with the output that is produced during compilation, are

```
*CMAP <CONST>
*STOPC
```

The first of the pair causes compile time output to appear on the stream specified by the given constant, and the last stops this output. The CMAP is a compile map giving line numbers and positions in store as various landmarks are passed. It is described more fully in 9.8.

To assist with understanding the action of the compiler, for example, if modifications are required, there is a further directive

```
*INFORM <CONST>
```

This will cause printing of various compiler lists and information depending on the value of the bit significant parameter (CONST) as follows.

Bit	Function
0	Print the itemised statement, every statement read
1	Print the text character by character as they are read
4	Print the analysis record of <COMPUTATIONS>.
5	Inhibit compile map

Bits 8 - 15 control MUTL output
(i.e. they are passed as bits 0 - 7 of the parameter to TL.PRINT)

9.4.3 INITIALISATION

If the MUSL compiler is not running in a full MUSS environment, in which the MUTL system can be initialised and stored by direct calls on MUTL procedures from the command stream of a compile run, facilities for this purpose are necessary in the compiler. In particular it is necessary to call TL and TL.END at the start and finish of the compile run, possibly make several calls on TL.SEG to establish the segments for the compilation in which case it will also be necessary to make calls on TL.LOAD and possibly to call TL.MODE for the compilation of individual modules. Thus the following directives are provided

```
*INIT<CONST>
*TLSEG<CONST><CONST><CONST>
*TLLOAD<CONST><CONST>
*TLMODE<CONST><CONST>
```

The parameter of *INIT is a bit significant CONST whose bits have the following meaning.

bit 0 = 0 the compilation is for a 32-bit machine
bit 0 = 1 the compilation is for a 16-bit machine
bit 1 = 1 the module following it the first of a compile run
bit 2 = 1 the module following is the last of a compile run.

If bit 1 = 1 a call will be made on TL and bits 8-15 of CONST will be used as the parameter of TL. If bit 2 = 1 there is no immediate action but the next *END encountered will finally result in a call on TL.END.

*TLSEG *TLLOAD and *TLMODE should be followed by the appropriate parameters separated by spaces.

9.4.4. LIBRARIES

Another directive, only relevent in a Compile and Load environment, is

```
*LIB[<NAME>!<NIL>]
```

The NAME should be the name of a library file. The action of the directive is to call the LIB command to open the library, and to automatically generate LSPECs (see 9.4) for the procedures it contains. If the name is omitted the action is to generate LSPECs for the system library.

9.4.5. BINARY PATCHING

There is also a *#<HEX.CONST> directive which allows binary machine instructions to be introduced into critical machine dependent sequences. This is used in MUSS, for example, to implement register dumping and reloading when a process change takes place.

9.4.6. MUSS LIBRARY CALLS

If the MUSL compiler is running in a MUSS environment, a call of any MUSS library procedure can be forced from the compiler by using in the MUSL source a MUSS command preceded by **. This facility is most often used to manipulate the input/output streams. For example, two separate files can be made to appear as one, to the compiler by using the 'IN' command in one file (i.e. '**IN name of file') at the point where the other file is to be included.

Another use of the facility is to make direct calls on the MUTL procedures that control the environment of the compilation.

The command following the ** up to the next newline is processed by MUSS not by the compiler and it should not therefore be terminated by ';' or a comment.

9.4.7 VSTORE TYPE

The VSTORE declarations of Section 9.5.10 introduce the idea of a default VSTORE type. This is normally the same as the default integer type but it may be reset by the directive

```
*VTYPE INTEGER32;
```

for example.

9.5 DECLARATIVE STATEMENTS

Every name used in a MUSL statement must be declared before it is used except for the reserved names of the language and label names. The declarations of the language are

```
<DECLARATIVE.STATEMENT> ::= <LABEL.DEC>!
                            <VAR.DEC>!
                            <PROC.DEC>!
                            <LIT.DEC>!
                            <DATA.VEC>!
                            <TYPE.DEC>!
                            <FIELD.DEC>!
                            <SPACE.DEC>!
                            <V.STORE.DEC>!
                            <IMPORT.DEC>
```

In effect a declaration specifies one or more names, defines the kind of entities to which they relate, and where relevent gives their properties. Thus a name declared in a LABEL.DEC is regarded as a label name and a name declared in a PROC.DEC is regarded as a procedure name. The former will have no additional properties but the latter might have properties concerned with the kind and type of its parameters and result.

9.5.1 LABEL DECLARATIONS

```
<LABEL.DEC> ::= <NAME>: ! <NAME> <>>:
```

A label declaration provides a way of referencing the statement which follows it in a '->'(goto) statement. Normally labels are automatically generated by Flocoder and the user will not be directly concerned with them. In effect the label name is a literal whose value is the address of the statement that follows it. Unlike all other declarations the scope of the name declared is the whole of the procedure or block in which it is declared. That is, forward references within a procedure or block are permitted, whereas in all other declarations, they are only effective onwards from the place where they appear.

If a label is to be referenced in a '->' statement of an enclosed procedure it must be declared using >:. When a '->' statement containing such a label is encountered the stack is reset to the frame that was established after entry to the procedure containing the label. All 'restart labels' of this kind must be declared before they are referenced in '->' statements.

9.5.2 VARIABLE DECLARATIONS

```
<VAR.DEC> ::= <TYPE><NAME.LIST>
```

A variable declaration specifies an arbitrary list of names of variables of a given type. They may be scalar or vector variables depending on whether or not the dimension is given in

```
<TYPE> ::= <SCALARTYPE>[<[><CONST><]>!<NIL>]
```

Scalar types may be numeric types, the names of user defined types (see later), pointers to scalar or vector instances of either of these, or pointers to procedures. In this latter case the name must be one for which at least a PSPEC and possibly a procedure definition has been previously given.

```
<SCALARTYPE> ::= <NUMERICTYPE>!<NAME>!
                 ADDR[<NUMERICTYPE>!<NAME>]!
                 ADDR<[>[<NUMERICTYPE>!<NAME>]<]>
<NUMERICTYPE>::= [INTEGER!REAL!LOGICAL]<SIZE>
<SIZE>       ::= 1!8!16!24!32!64!128!<NIL>
```

For example

INTEGER32	denotes a 32-bit signed integer
LOGICAL8	denotes an 8-bit unsigned integer
INTEGER32[10]	denotes a vector of 10 32-bit integers
ADDR INTEGER32[10]	denotes a vector of 10 addresses of 32-bit integers
ADDR [INTEGER32]	denotes the address of any vector of 32-bit integers
ADDR P1	can be used to indicate the address of a particular type of procedure where P1 is a name defined in a PSPEC

The main numeric types are integer and real, but each may exist in a range of sizes. For type INTEGER the size may be 8, 16, 24 or 32, short operands will be sign extended up to the size of the COMPUTATION in which they appear. If the size indication is omitted the natural size (16/32 bits) of the target machine will be chosen. Type REAL may only be 32, 64 or 128.

The third numeric type (LOGICAL) is similar to INTEGER, but the sign propagation of short operands is suppressed and sizes 1 and 64 are also permitted. It should be noted that LOGICAL is merely a name for unsigned integers. Entities of this type can be used in any INTEGER context. There is no separate LOGICAL mode of arithmetic but the logical operators representing 'and', 'or' and 'non-equivalence' can be used in INTEGER computations.

In the declaration of vectors the CONST should have an integer value >0. It specifies the number of elements in the vector and subsequent accesses should regard them as being numbered 0 to CONST-1.

Variables which are declared inside a procedure are local to that procedure and are allocated space on its run time stack frame. The variables of a procedure declared at the outer level of a module can be accessed non-locally from any nested procedure. Variables declared at the outer level of a module are global to all the procedures of the module. More general non-local access is not permitted. The positions occupied by global variables depend upon the use of the *GLOBAL directive (9.4).

9.5.3 PROCEDURE DECLARATIONS

```
<PROC.DEC> ::= [PSPEC!LSPEC]<NAME>[<PSPEC>!=<NAME>]
<PSPEC> ::= ([<T.LIST>!<NIL>])[/<SCALARTYPE>!<NIL>]
```

A declaration must be given for every procedure, before it is used or defined. This declaration defines the NAME to be a procedure name indicates whether the procedure is to be given (PSPEC) or found in a library (LSPEC), gives the type of each parameter, and the type of the result (after the /), if it is a function procedure. Only scalar types are permitted as parameters, and results are further restricted to non user defined types and addresses of user defined types.

```
<T.LIST>  ::= <SCALARTYPE>[,<T.LIST>!<NIL>]
```

Thus vectors cannot be passed as parameters but their addresses can. Also addresses of user defined types can be passed thereby enabling procedures to place results in the associated variables.

If a group of procedures have identical specifications a full specification must be given for the first then the '='s option can be used in all further PSPECs.

Sometimes it might be appropriate to give a specification for a name of a class of procedures (e.g. TRIGFN) which is not itself a procedure, and to use it to declare the procedures of that class, for example

```
PSPEC SIN = TRIGFN
```

It may also be used to declare pointers to procedures of that particular class, for example

```
ADDR TRIGFN
```

In simple Compile and Load cases *LIB (see 9.3) would be used in preference to LSPECs. LSPECs are intended for cross compiling situations where the resolution of the location of the specified procedure has to be delayed until eventual load time and is done in relation to the procedures available in the target machine rather than the compiling machine.

9.5.4 LITERAL DECLARATIONS

```
<LIT.DEC>   ::= LITERAL[/<SCALARTYPE>!<NIL>]<N.L.LIST>
<N.L.LIST> ::= <NAME>=[<LITERAL>[,<N.L.LIST>!<NIL>]!<NIL>]
```

These declarations provide a facility to define names, as literal names having compile time values (constants). When a NAME associated with a literal value is used (for example, as a <CONST>) the literal value is substituted in its place. Several literals can be defined in the same declaration providing they are all of the same TYPE. If no TYPE is given INTEGER (default size) is assumed.

```
<LITERAL> ::= <CONST.EXPR>!<AGGREGATE.CONST>
<CONST.EXPR> ::= <CONST>[<CONST.OP><CONST.EXPR>!<NIL>]
 <CONST.OP> ::= + ! - ! & ! <!> ! * ! /
 <AGGREGATE.CONST> ::= <CONST>[\<AGGREGATE.CONST>!<NIL>]
```

Normally literals will be single constants. However, some literals may be functions of other (earlier defined) literals hence a simple form of expression is allowed. It can only be used with literals of type INTEGER and LOGICAL hence the forms of CONST used must also be of these types. The evaluation is left to right and the meaning of the operators is

+ meaning add
- meaning subtract
& meaning logical 'and'
! meaning logical 'or'
* meaning multiply
/ meaning divide.

Only the NIL form of expression (i.e. part of <N.L.LIST> following =) is permitted when the TYPE is a pointer type, and it generates a NIL pointer of the given type. Aggregate literals correspond with user defined types, and each of their fields is defined by a separate constant. Some examples of literal declarations are

```
LITERAL/INTEGER     COUNT=10, SIZE=COUNT+1;
LITERAL/IPAIR       I1=100\3;
LITERAL/ADDR[$LO8] NIL.AL=;
```

IPAIR is defined in the examples of type declarations given later.

In practice literal declarations have sometimes been very long with many names declared in the same statement. Thus the compiler has been organised to take them a line at a time. As a consequence of this each intermediate line must end with ','. For example

```
LITERAL A = 1,
        B = 2;
```

is acceptable but

```
LITERAL A = 1, B
          = 2;
```

is not acceptable.

9.5.5 DATA VECTORS

A Data Vector is a one dimensional array of literals having a name which can be used like any other vector name in order to access the elements of the vector as operands. However, if the data vector is stored in a read only segment it cannot be altered by the program. If it appears at the outer level of a MODULE it will be placed with the global variables of the module and if it is contained within a procedure it will be stored with the code. Thus the placement of data vectors can be determined by the programmer by preceding them with *GLOBAL directives in the first case and *CODE directives in the second case.

```
<DATAVEC>  ::= DATAVEC<NAME>(<SCALARTYPE>)
                <LITERALS>
                END
```

Any number of LITERALs are allowed but they must all be of the same given TYPE. The only permitted pointer type of data vector element is the address of a procedure or data vector (ADDR<NAME>).

```
<LITERALS>  ::= <LITERAL>[<[><CONST><]>!<NIL>]
                 [<LITERALS>!<NIL>]
```

Obviously the individual literals must be separated and it is usual to use space or newline for this purpose. If a LITERAL is followed by a bracketed <CONST> it will be repeated the CONST number of times. Only CONSTs with integer values >0 should be used in this context. For example:

```
DATAVEC NAME($LO8)
'J 'A 'C 'K
END;
```

is a VEC called NAME of 4 bytes,and

```
LITERAL COUNT = 4
```

```
DATAVEC CONSTS($IN)
1  2  3
10[5]
20[COUNT]
6
END;
```

is a VEC of 13 integers.

Because of the high incidence of character data vectors in some parts of MUSS, for example, as error messages, a special construct is provided to abbreviate lists of character literals. It is "<CHSTRING>". Thus the first example above could be written

```
DATAVEC NAME($LO8)
"JACK"
END;
```

9.5.6 TYPE DECLARATIONS

```
<TYPE.DEC>   ::= TYPE<NAME> IS <STRUCTURE>
<STRUCTURE>  ::= <FIELDS>[OR<STRUCTURE>!<NIL>]
<FIELDS>     ::= <TYPE><N.LIST>[<FIELDS>!<NIL>]
```

Type declarations allow the user to declare a NAME as a type name and to associate it with a STRUCTURE comprising several fields of simpler types. The fields are given names so that they may be subsequently accessed as operands. Consecutive fields of the same type are declared by giving a list of names after the TYPE. If fields are of different types a TYPE word should appear before each field name. A TYPE.DEC which has an alternative is called a UNION. In all declared instances of a UNION, the compiler will allow space for the largest alternative form, but the onus is on the user to know which alternative is present at any point in time.

9.5.7 EXAMPLES OF VARIABLE/TYPE DECLARATIONS

The simplest example of a declaration is where no dimension is given, e.g.

```
INTEGER I,J;
```

In this case I and J will be local variables of type INTEGER. If a vector of elements of type INTEGER is required a dimension must be given thus:

```
INTEGER[25]VECI;
```

In this case the 25 INTEGER elements of VECI will be numbered 0,1,...24.

New types are created by use of the TYPE declaration, e.g:

```
TYPE IPAIR IS $IN PROP1, PROP2;
TYPE CONDITIONS IS REAL TEMP, PRESS, VOL;
```

These TYPE names can be used to declare scalar or vector instances of the TYPEs in question, e.g:

```
IPAIR I1,I2,I3;
```

declares three integer pairs and

```
CONDITIONS [20]U,V;
```

declares two vectors each having 20 elements containing a TEMP PRESS and VOL component.

A TYPE may have several different fields, e.g:

```
TYPE IOAREA IS $IN SPN, PID $LO8[100]MBUFF
```

declares a TYPE IOAREA with three fields which are the two integers SPN, PID and a vector (MBUFF) of 100 bytes. This might be used to declare an IOCONT vector containing 16 IOAREA's thus:-

```
IOAREA[16] IOCONT;
```

It should be noted that every field has a name which must be unique within the type the purpose of which is to provide a means of accessing it as described in 9.5.

9.5.8 FIELD DECLARATIONS

These are duplicate declarations for the fields of structures which allow them to be 'selected' and subsequently accessed by their field name only. The syntax of the statement is given below, but its meaning and use is discussed in 9.6.

```
<FIELD.DEC>  ::=  SELECT<CONTEXT>
```

where CONTEXT is defined in 9.6.

9.5.9 SPACE DECLARATION

It is convenient to be able to reserve some unmapped store to be dynamically mapped by means of the MAKE function. The SPACE.DEC provides for this.

```
<SPACE.DEC> ::= SPACE<NAME>[<[><CONST><]>!<NIL>]
```

If these declarations appear inside a procedure the space will be reserved on the run time stack otherwise it will be reserved in the current global area (i.e. that last selected by a *GLOBAL directive).

The NAME may subsequently be used as the parameter of a call for the the built-in function MAKE. The CONST specifies the required amount of space in bytes and must be an integer >0. Thus a space of 1 Kbytes called HEAP1 would be declared

```
SPACE HEAP1[1024]
```

The use of the MAKE function and the provision made for 'garbage collection' in such spaces is discussed in 9.6.5

In some situations in MUSS areas of store arise dynamically. For example, when a file is opened, or a message is received. These can be treated as 'SPACE's if a space name declared without parameters is subsequently associated with an area by means of the built in function POSN also described in 9.6.5.

9.5.10 V-STORE DECLARATIONS

The MUSS 'ideal machine' is defined as a set of 'ideal V-lines' concerned with the control of peripherals, store management hardware and CPU status. V.STORE.DEC is concerned with their mapping into an actual machine. There are two forms of V.STORE.DEC as follows

```
<V.STORE.DEC> ::= VSTORE <VDEFS>
      <VDEFS> ::= <VDEF>[,<VDEFS>!<NIL>]
      <VDEF>  ::= <NAME>[<[><CONST><]>!<NIL>]
                  [%<HEXCONST>!<NAME>]
                  [<<><NAME>!<NIL>]
                  [<>><NAME>!<NIL>]
```

One option in V.STORE.DEC provides a means for associating an absolute address with an ideal V-line. It uses the CONST option where the CONST must have an integer value giving the address of the V-line in bytes and its type will be taken to be the default VTYPE (Section 9.4.7). For example

```
VSTORE LPTSTATUS %3FF4C;
```

associates LPTSTATUS with the byte address %3FF4C. In cases where the machine VSTORE does not correspond to the ideal machine VSTORE, it is necessary to map the ideal VSTORE onto a variable and to emulate the special actions of the ideal VSTORE by using PREPROCs and POSTPROCs. Thus another form of VDEC is

```
VSTORE V1 PSEUDOV1 <PROC1 >PROC2
```

Here the VSTORE variable V1 is mapped onto the variable PSEUDOV1 (and takes its type), and the names which (optionally) follow are the names of procedures. If the first procedure name is present (e.g. PROC1 above) it will be called before a read access on V1 and the second (PROC2) will be called after a write access to V1. The TYPE of VSTORE variables may be machine dependent but it will usually be type INTEGER, although if a VSTORE variable is mapped into another variable it takes the TYPE of that variable.

Some VSTORE entities, for example, page address registers occur naturally as vectors. In these cases a dimension should be given as for vectors of ordinary variables.

A procedure used as a PREPROC or POSTPROC is a restricted form of procedure. It may not have local variables, parameters or a result. However, the code it contains may use the reserved name VSUB to obtain the subscript in the case where it relates to a vector of V-lines. In effect the VSTORE declaration serves as the 'PSPEC' of the 'PROC's hence it should precede the PROCs and be at the same block level. In particular MUTL implementations, the restrictions may be more severe (e.g. language constructs that cause the compiler to generate local variables must be avoided).

9.5.11 IMPORT DECLARATIONS

A module sometimes uses imported entities without requiring a full specification to precede the module heading. This applies to TYPEs, VSTORE, LABELS and LITERALs. Thus the following special IMPORT statement is provided

```
                <IMPORT.DEC> ::= IMPORT<KIND><IMP.LIST>
                <KIND>       ::= VSTORE[<NUMERICTYPE>!<NIL>]!LITERAL!
                                 TYPE!LABEL[<>>!<NIL>]
                <IMP.LIST>   ::=
<NAME>[<[><]>!<NIL>][,<IMP.LIST>!<NIL>]
```

A type may be specified for an imported VSTORE and the NAME[] construct is used to indicate when a vector instance of VSTORE is to be imported. In the case of LABEL a following '>' will cause them to be treated as 'restart' labels.

If a type name is imported in the above manner only the type name and not its fields are accessible in the module into which it is imported. A full duplicate type definition must be given before the module heading if use is to be made of the field names of the type. Similarly if procedures or variables are imported, full duplicate definitions are necessary. Clearly any entity that is imported into a module must be exported from another module, and modules related in this way must be loaded in the same load run, or compiled in the same compile run if a generate executable binary option is used.

9.6 OPERAND FORMS

```
 <OPERAND>      ::=  [^!<NIL>]<VARIABLE>[OF<CONTEXT>!<NIL>]!
                     <CONST>!
                     <NAME>[^!<NIL>]([<P.LIST>!<NIL>])!
                     (<COMPUTATION>)!
                     (<COND.COMP>)!
                     <BUILT.IN.FUNCTION>
<VARIABLE>      ::=  <NAME><SUBSCRIPT>[^<SUBSCRIPT>!<NIL>]
<CONTEXT>       ::=  <VARIABLE>[OF<CONTEXT>!<NIL>]
<SUBSCRIPT>     ::=  <[><COMPUTATION><]>!<NIL>
<P.LIST>        ::=  <COMPUTATION>[,<P.LIST>!<NIL>]
<COND.COMP>     ::=  IF<CONDITION>THEN<COMPUTATION>
                     ELSE<COMPUTATION>
```

<COMPUTATION> and <CONDITION> are defined in 9.6.

The operand capability of MUSL is moderately complicated and warrants some discussion. Inevitably the examples given reflect the structure of the COMPUTATIONs described in the next Section but their meaning should be self evident.

9.6.1 VARIABLES

Considering first the case where no CONTEXT is given, VARIABLES which are scalars are represented by names, which are declared as described in 9.4. They may be Local, Non-local or Global. Those which are vector elements are represented by the vector name followed by a subscript. The subscript (or index) is the result of a computation which gives the required element number. For example, A[0] denotes the first element of a vector A and A[9] denotes its 10th element. It follows that A[I-1] denotes the Ith element. If a variable contains the address of the required variable it must be followed by "^" to explicitly dereference it. Any variable so dereferenced might have been a pointer to a vector in which case a further subscript is required. For example if the NAME in the definition of VARIABLE refers to the address of a vector, for example, it has been declared

```
ADDR[INTEGER] name1;
```

and the address of a suitable vector has been assigned to name1, then an element in the vector is accessed thus:

```
name1^ [subscript]
```

Whereas if it were the NAME of a vector of addresses of integers, declared as

```
ADDR INTEGER[10] name2;
```

any one of the integers addressed could be accessed thus:

```
name2 [subscript]^
```

If a reference to a variable is to be created then its name is preceded by an "^". Thus

```
^name1^[4] => name2[3]
```

computes the address of the 4th element of the vector of integers whose address is given by name1 and assigns it to the 3rd element of the vector of addresses name2.

When a variable is a field of an aggregate type its CONTEXT must also be given. E.g.

```
SPN OF IOCONT[I]
```

If it is required to access an element of a vector which is itself a field in an element of a vector, for example an element of the MBUFF defined in 9.5, the notation is:-

```
MBUFF[J] OF IOCONT[I]
```

meaning the Jth element of the MBUFF in the Ith (aggregate) element of IOCONT.

In general aggregate types can be nested to an arbitrary depth, hence, CONTEXT is defined recursively. It should be noted that the last name given in the specification of a variable refers to an instance of a declared object, whereas the preceding names are field names. Any of these names might be subscripted and/or dereferenced.

The repetition of the full specification, when a series of operations are performed on a variable embedded in a STRUCTURE, is tedious. The FIELD.DEC described in 9.6.3 allows them to be accessed through their field names only.

9.6.2 CONSTANTS

These were described in 9.3. Any of the usual forms are permitted but they must be type compatible with the context in which they appear (see 9.7.2).

9.6.3 FIELD DECLARATIONS

The syntax was given in 9.4. Its use is illustrated in the example below.

```
        SELECT IOCONT[I]
```

serves two functions. It causes the address of the element of IOCONT specified by I to be noted and it implicitly declares all the fields of IOCONT to be accessible by their field name only.

Given this, and assuming SPN, PID and MBUFF are fields of IOCONT, a statement such as

```
        IF SPN OF IOCONT[I] = 0 THEN
        FOR J<100 DO 0 =>  MBUFF[J] OF IOCONT[I]OD
        FI
```

could be rewritten

```
        IF SPN = 0 THEN FOR J<100 DO 0 => MBUFF [J]OD
```

The addresses of the selected fields are computed at the time the declaration is processed. Hence, subsequent changes in the values of variables which appear as subscripts of the selected fields will be ineffective.

9.6.4 PROCEDURE CALLS

Procedures are called by giving their name (or a dereferenced pointer to a procedure) followed by a P.LIST enclosed in brackets. The parameters must be COMPUTATIONs which yield values of the correct type, although the usual automatic type conversions that apply in COMPUTATIONs may be assumed. For example, an INTEGER can be substituted for a REAL. Type and type conversions are discussed in 9.7.2.

Procedures which return results do so by assigning them to the name of the PROC as described earlier.

9.6.5 BUILT-IN FUNCTIONS

Several built-in functions have already been mentioned but the significance of "built-in" has not been explained. Their textual representation is that of a procedure. They are built into the compiler because their implementation uses facilities not available at the user level. These fall into three main groups.

direct manipulation of addresses
variable TYPE parameters

interpretive entry to libraries.

A complete list of the built-in functions is

```
        MAKE(ANY TYPE,$IN,[ANY SPACE!A BYTE ADDRESS!<NIL>]
                                                  )ADDR OF GIVEN TYPE
        PART(ADDR OF A VEC,$IN,$IN)ADDR(OF A VEC OF LIKE TYPE)
        SIZE(ADDR OF A VEC)$IN
        POSN(ANY SPACE,A BYTE ADDRESS)
        BYTE(<COMPUTATION>)$IN(A BYTE ADDRESS)
        LLST()
        LPST(ANY BASIC TYPE, COMPUTATION)
        LENT(INTEGER COMPUTATION,ANY BASIC TYPE)GIVEN BASIC TYPE
```

9.6.5.1 Dynamic Allocation of Store

The built-in function MAKE exists to allocate space at runtime for an object in a space previously created by a SPACE declaration. The function has three parameters. The first is a type name, and the second is a number. Space for the specified number of objects of the given type is reserved. The third parameter of MAKE specifies the name of the SPACE that is to be used.

The function yields a value of type ADDR of type of object created. If the number of objects required is the constant 0 a pointer to a scalar is yielded, otherwise a pointer to a vector instance of the objects is yielded.

Thus:-

```
      MAKE(REAL,0,HEAP1)
```

generates space for a REAL value in the space specified as HEAP1 and yields its address.

This implies that associated with HEAP1 is an index that keeps track of the amount of space actually in use. Obviously the need might arise to note and reset this index and this is achieved by using the name of the space as an integer variable.

In fact the third parameter of MAKE can be omitted and in this case the run-time stack frame of the procedure will be used, or it can be an explicit byte address.

9.6.5.2 Address Manipulation

Clearly operands have addresses but normally the programmer is only concerned with their names or with the names of ADDRESS type variables. Addresses may be generated or dereferenced by means of the "^" operator described earlier. However, certain exceptional requirements arise in system programming which make it necessary to manipulate addresses more directly. A simple example occurs when a character string has to be read, stored and returned as a result. In this case a global or parametric byte vector of adequate length could be used to store the characters and the built-in-function:

PART(<COMPUTATION>,<COMPUTATION>,<COMPUTATION>)

is available to generate the appropriate result to return. The first COMPUTATION should yield the address of a vector and the next two COMPUTATIONs give the first and last subscripts of the partition whose address is required. As the parameters may be evaluated in an implementation dependent order they should have no side effects which would affect the values of the other parameters.

Again in the context of parametric vectors it might be necessary to discover the number of elements that it contains. The function

SIZE(<COMPUTATION>)

yields the size of the vector whose address is yielded by the COMPUTATION.

Another requirement arises as a result of the Operating System allowing segments to pass from one virtual machine to another, either as files or messages. This means that a program can acquire information at a particular address unknown to the compiler. In this case usable addresses for the objects can be obtained from the built-in function MAKE providing the area of store can be treated as a SPACE. The built-in function

POSN(<SPACENAME>,<ADDRESS>)

associates the given SPACE name with the area of store whose byte address is given as the second parameter. The first parameter is the name of the SPACE and the second is its address in byte units. Alternatively, for similar reasons a byte address can be used directly in MAKE. Byte address of variables can be obtained by using the function

BYTE(<COMPUTATION>)

The COMPUTATION must yield a result of pointer type.

9.6.5.3 Interpretive Library Calls

These facilities are required by, for example, a command language interpreter. A normal (compiled) procedure call is broken down by a compiler into several steps. These are notionally

```
prepare to call a procedure
stack the first parameter
stack the second parameter
              .
              .
stack the last parameter
enter the procedure
assign result
```

Further code might be interleaved with these to compute the parameters. In order to facilitate interpretive procedure calls these same steps are made available individually as built-in functions. They are

```
LLST()
LPST(TYPE,COMPUTATION)
LENT(procedure index, TYPE)
```

The procedure index is the integer result returned by the library procedure FINDN. The LENT procedure is used to enter the procedure indicated by the FINDN index given to it. The TYPE (or 0) is the type of the result of the given procedure and if given then this call on LENT may be followed by an assignment to a variable of that type. Two other related library procedures are usually required in order to make an interpretive procedure call. They are PARAMS and PARAMTYPE.

9.7 IMPERATIVE STATEMENTS

The Imperative Statements include both:

Computational Statements
and Control Statements

Computational Statements may be conditional, unconditional, repeated WHILE some CONDITION holds or FOR a given number of times (any of which may be zero times), thus:

```
<IMPERATIVE.STATEMENT> ::= <COMP.ST>!<CONTROL.ST>
<COMP.ST>           ::= <COMPUTATION>!
                     <IF.ST>!
                     <WHILE.ST>!
                     <FOR.ST>
<CONTROL.ST>        ::=  <GO.ST>!
                     <SWITCH.ST>!
                     <ALT.ST>!
                     EXIT
<IF.ST>            ::= IF<CONDITION><ACTION>
<WHILE.ST>         ::= WHILE<CONDITION>
                       DO<STATEMENTS>OD
<FOR.ST>           ::= FOR[<NAME><<>!<NIL>]<COMPUTATION>DO
                       <STATEMENTS>OD
<ACTION> ::=           ,<GOST>!THEN<STATEMENTS><ELSECL>FI
<ELSECL>       ::=    ELSE<STATEMENTS>!<NIL>
```

CONDITIONS also appear in conditional control statements and are described in 9.7.3. In the WHILE statements the CONDITION is repeatedly evaluated. Each time it yields a true result, the imperative statements between the following DO and OD are obeyed. On the first occasion that the condition yields false, control passes to the next statement after the OD. The FOR statement provides for a similar set of imperative statements to be executed a given number of times as specified by the control COMPUTATION which is evaluated once at the start. As the repetition proceeds a counter is maintained which starts from zero. When it reaches the specified value the repetition stops. If this "control variable" is to be used during or after the loop an INTEGER variable name must be given in the NAME. On completion of a FOR loop the value of the 'control variable' will be the value of the control COMPUTATION.

9.7.1 COMPUTATIONS

A computation is expressed as a sequence of operators and operands thus:

```
<COMPUTATION> ::= <OPR.OPD.SEQ>
```

```
<OPR.OPD.SEQ> ::= <OPERAND>[<OPERATOR><OPR.OPD.SEQ>!<NIL>]
```

Every COMPUTATION is an expression and the final result which occurs after complete evaluation of the OPERATOR OPERAND sequence (from left to right) is the VALUE of the expression. This value is discarded if the computation is a statement in its own right.

The OPERATORs are the following

+	meaning	add
-	meaning	subtract
*	meaning	multiply
/	meaning	divide (rounding towards 0 for integer result)
&	meaning	logical "AND"
!	meaning	logical "OR"
-=	meaning	logical "NON-EQUIVALENCE" (or exclusive OR)
-:	meaning	reverse subtract
/:	meaning	reverse divide
+>	meaning	ADD into store
->	meaning	SUBTRACT from store
*>	meaning	MULTIPLY into store
/>	meaning	DIVIDE into store
&>	meaning	AND into store
!>	meaning	OR into store
-=>	meaning	NONEQUIVALENCE into store
=>	meaning	assign to
-:>	meaning	reverse subtract from store
/:>	meaning	reverse divide into store
<<-	meaning	left logical shift
->>	meaning	right logical shift

When an OPERATOR is followed by ">" (excepting of course ->>,<<- and =>), first the OPERATOR is applied (reversed if not commutative) to the current result and the following OPERAND, then this new result is assigned to the following OPERAND and finally the new result is carried forward to the next stage of the COMPUTATION. For example, 1->X will decrement X and generate a result "X-1". Note that some operators namely &, !, -=, &>, !>, -=>, <<-, ->> are not available in floating point mode.

General computations are not permitted on variables of aggregate or pointer types. They may, however, be moved and compared, but in the case of aggregates not both in the same statement. For example:

```
AGG1 => AGG2 => AGG3
$IF AGG1 /= AGG2 /= AGG3 $TH
```

are acceptable.

```
$IF AGG1 /= AGG2 => AGG3 $TH
```

is not acceptable.

9.7.2 TYPE

The TYPE of arithmetic used in evaluating a computation is determined by its operands. In any COMPUTATION (or COMPARISON) the TYPE and size of the operands must be compatible. Obviously if they are all the same, they are compatible, but there will be automatic type and size conversion in some other cases.

For any computation the compiler determines a computation type by looking ahead up to and including the first 'assigned to' operand, or the end of the computation if this occurs first. The operand of the largest type determines the computation type. The possible computation types are INTEGER (size 16/32/64), REAL (size 32/64/128), user defined types and the permissible address types (any REAL type is considered larger than any INTEGER type). However, the operations available are in some case restricted as shown in the table below (/ indicates allowed x indicates not allowed).

	REAL(32/64/128)	INTEGER(16/32)	INTEGER(64)	USER/ADDR TYPES
+	/	/	x	x
-	/	/	x	x
*	/	/	x	x
/	/	/	x	x
&	x	/	/	x
!	x	/	/	x
-=	x	/	/	x
-:	/	/	x	x
/:	/	/	x	x
+>	/	/	x	x
->	/	/	x	x
*>	/	/	x	x
/>	/	/	x	x
&>	x	/	/	x
!>	x	/	/	x
-=>	x	/	/	x
=>	/	/	/	/
-:>	/	/	x	x
/:>	/	/	x	x
<<-	x	/	/	x
->>	x	/	/	x

Type conversions are permitted only as follows

OPERAND	COMPUTATION TYPE							
TYPE	INTEGER	16	32	64	REAL	32	64	128
LOGICAL1		/	/	/		/	/	/
LOGICAL8		/	/	/		/	/	/
LOGICAL16		/	/	/		/	/	/
LOGICAL24		/	/	/		x	x	x
LOGICAL32		/	/	/		/	/	/
LOGICAL64		/	/	/		/	/	/
INTEGER8		/	/	/		/	/	/
INTEGER16		/	/	/		/	/	/
INTEGER24		/*	/	/		/	/	/
INTEGER32		/*	/	/		/	/	/
REAL32		/*	/*	/*		/	/	/
REAL64		/*	/*	/*		/	/	/

The conversions marked '/*' are permitted but they may generate arithmetic overflow.

If an 'assigned to' operand is less than the type of the computation, a type conversion will be applied, after the computation has been evaluated. This type converted result is the value carried forward as the first operand when further operators and operands follow. If the expression contains further assignment this process is repeated for each. For example, given

```
INTEGER32    I,J
LOGICAL64    LL
REAL32       R1
REAL64       RR2

I+J-1=>R1   will be evaluated in 32-bit real mode
I+J+R1=>RR2 will be evaluated in 64-bit real mode
R1*RR2=>J   will be evaluated in 64-bit real mode
            but converted to 32-bit integer mode
I<<-8!J<<-8=>LL will be evaluated in 64-bit integer mode
```

Parenthesised computations yield an operand of the 'final' type used inside the parentheses. The result is then converted, if necessary to the type of the expression that contains it. Thus in

```
(I+J-1) => R1
```

the arithmetic would occur in INTEGER32 mode and the result would be converted to REAL32 before being assigned to R1.

In a TEST involving COMPARISONs (see below) each computation is evaluated in the type of the 'largest' computation. For example

```
CH()<<-8!CH()<<-8!CH()<<-8!CH()<<-8!CH()<<-8 = NAME
```

will be evaluated in INTEGER64 mode even if CH delivers a LOGICAL8 result providing NAME is LOGICAL64.

Parameter expressions are evaluated in a type of arithmetic consistent with the expression being followed by an assignment to a variable of the type of the formal parameter.

9.7.3 CONDITIONS

```
<CONDITION> ::= <TEST>[<LOGOP><CONDITION>!<NIL>]
<TEST> ::=          <[><CONDITION><]>!<COMPUTATION><COMPARISON>
<LOGOP> ::=         OR!AND
```

The simplest form of CONDITION is a TEST in the form of a COMPARISON applied to the result of a COMPUTATION. The syntax for COMPARISON is:

```
<COMPARISON>        ::= <COMPARATOR><COMPUTATION>
                    [<COMPARISON>!<NIL>]
<COMPARATOR>        ::= =!/=!<<>!<>>!=<<>!<>>=
```

This allows several COMPARISONS to be made against the same RESULT. They must all be satisfied for the CONDITION to be satisfied.

Examples of simple conditions are:

```
X = Y
X-1 /= Y/Z
INSYM() =<'Z >='A(meaning the same as below)
```

More complicated conditions would use AND and OR, e.g.

```
INSYM() => SYM =< 'Z AND SYM >= 'A
X=Y AND Z=10 OR P /= 4
```

The AND symbol has greater binding power than OR hence the following parenthesis is implied:

```
[X=Y AND Z=10] OR P /= 4
```

Conditions are evaluated left to right and the evaluation ceases when the status of the condition is known. This is when a test that succeeds is followed by "OR" or one that fails is followed by "AND".

9.7.4 CONTROL STATEMENTS

```
<GO.ST>        ::= -><NAME>
```

Any basic statement in a program can be labelled thus defining a label name. A GOTO statement may reference these label names or

OPERANDS which are of type LABEL. Usually, however, the labels and '->'s are generated by Flocoder not the programmer. Non local '->'s are permitted only where the name has already been appropriately declared (see 9.3). They are normally concerned with error recovery.

The switch statement also uses labels. Its syntax is

```
<SWITCH.ST> ::= SWITCH <COMPUTATION>\<NLIST>;
```

Here the value of the computation decides which NAME is selected in the N.LIST (value 0 selecting the first), and a GOTO that NAME is obeyed. The NAMEs must be locally defined LABELs.

An alternative statement specifies a COMPUTATION and a list of STATEMENTS thus:

```
<ALT.ST> ::= ALTERNATIVE <COMPUTATION>FROM
             <STATEMENTS>
             END
```

9.7.5 IMPLICIT BLOCKS

Blocks were introduced in 9.2.6 and the scope restrictions they impose in 9.2.7. Some groups of STATEMENTS are treated as if they constitute a block even though explicit BEGINs and ENDs are not given. They are the STATEMENTS used in

```
    IF.ST
    WHILE.ST
    FOR.ST
and ALT.ST
```

One important effect of this is that GOTOs entering or leaving such statement groups are not permitted.

9.7.6 EXAMPLE OF A PROCEDURE DEFINITION

As an illustration of a complete sequence of MUSL statements, consider the following procedure for reading as decimal integer

Its specification is

```
PSPEC IN.I()/INTEGER;
PROC IN.I;
INTEGER SIGN, CH, ANS;

WHILE IN.CH() => CH =< " " DO OD::IGNORE SPACES

IF CH = "-" THEN::DEAL WITH OPTIONAL SIGN
```

```
   1 => SIGN;
   IN.CH() => CH;
ELSE
   0 => SIGN;
   IF CH = "+" THEN
      IN.CH() => CH;
   FI
FI

IF CH - "0" => ANS >= 0 =< 9 THEN
   WHILE IN.CH() - "0" => CH >= 0 =< 9 DO
      ANS * 10 + CH => ANS;
   OD

   IN.BACKSPACE (1);
   IF SIGN = 0 THEN
      ANS => IN.I;
   ELSE
      0 - ANS => IN.I;
   FI

ELSE
   ENTER.TRAP (6,8);
FI

END
```

9.8 OPERATIONAL CHARACTERISTS OF THE COMPILER

CHAPTER 10 FORTRAN 77

10.1 INTRODUCTION

The MUSS FORTRAN compiler will implement Fortran 77 as defined in the document 'BSR X3.9 FORTRAN 77 dpANS FORTRAN Language X353/90' of the American National Standards. The compiler attempts to remain close to the standards defined, any deviations from standard are described below.

10.2 NON-STANDARD FEATURES

Holleriths have been included as an extension to the standard language. The use of Hollerith constants is restricted to DATA and CALL statements in a manner described in section 21, appendix C, of the Fortran 77 standard. In this implementation a maximum of four Hollerith Characters can be contained in one storage unit.

The standrad Fortran character set has been extended to include the tab and EOT characters. A Fortran source line may include a tab character. A tab in columns 1 to 5 of the Fortran source line has the affect of sufficient spaces to ensure the following character is treated as being in column 6. Tabs in column 6 and beyond are treated as spaces. The EOT character is used to mark the end of the Fortran source program and is an alternative to the *END directive (see 10.5).

MUSS commands may be included in the Fortran program source for optional interpretation at compile time as they are encountered. The option to interpret the MUSS commands is activated by the second Parameter to the compiler, as described in section 10.8. MUSS commands to be interpreted at compile time consist of an asterisk in columns one and two of an input card followed by an alphabetic character which begins the command, any other format is treated as a comment card begining with an asterisk and is ignored. If the option is not switched on the directives are treated as normal comments.

MUSS commands can occur anywhere a Fortran comment could occur.

Compiler directives (which are described in Section 10.5) are also non-standard extensions to the Fortran standard language.

This compiler is more generous than the Standard in allowing any comments after the last END of the last program unit (and before any *END directive). Any program which has comments (or even MUSS commands) beyond the last END would be non-standard, but would be acceptable to this implementaion.

10.3 PROGRAM LAYOUT

The sub-program units making up the complete Fortran Program may be compiled in any order. The lines of Fortran should conform to the Standard which describes the conventions for comments, continuations and the significance of the columns. The Fortran character set has been extended to allow tabs to column 6 as previously mentioned. The end of the compilation should be marked by a *END directive (see 10.5) or the EOT character.

10.4 FAULT MONITORING

10.4.1 Compile Time Faults

Faults at compile time are divided into warnings and fatal errors. A program containing only warnings may be run, as the warnings only refer to non-standard or bad features of the program (such as GOTO the next statement). Fatal errors are issued when the compiler is unable to understand the supplied program and can generate no code for the offending statement. Each faulty line is output by the compiler followed by the fault messages.

Every fault message will attempt to locate the fault within a statement by either an upward arrow below the point the compiler reached on determining the fault or including an offending name from the statement in the message. The former method is usually used for syntactic faults, the latter for semantic faults. At the end of a compilation the total number of message faults are printed.

10.4.2 Run Time Faults

Faults at run time will cause MUSS to trap the program, and these traps will be handled as any other high level language trap. A fault message usually accompanies such a trap. The Fortran run-time system uses trap number 6 for any input/output faults. A list of Fortran trap 6 reasons is given below.

101 Inconsistant field descriptor for input/output list item.
102 Illegal character in list directed complex character.
103 Illegal use of null value in list directed complex constant.
104 Attempted read beyond end of record.
105 No field descriptor for input/output list item.
106 Illegal character in integer or exponent.
107 Illegal value separator.
108 Illegal use of repeat counts.
109 Zero repeat count not allowed.
110 As 104.
111 Illegal character in logical item.
112 Illegal character (in repeat count?).
113 * missing from a repeat count.
114 Illegal character in a real.
115 Illegal sign in integer or exponent.
116 Attempted write beyond end of record.
117 Illegal carriage control char on output.
118 Illegal run time format.
119 Format label specified not defined.
120 No digit following sign.
121 Reading beyond sequential ENDFILE record.
122 Illegal unit access.
123 Invalid parameter in OPEN.
124 Invalid parameter in CLOSE.
125 Writing Direct Access record of wrong length.
126 Writing beyond sequential record.
127 Invalid unit number.
128 Too many units connected.
129 Invalid Fortran file format.

10.5 COMPILER DIRECTIVES

Compiler directives are Fortran statements (and therefore start after column 6) which commence with an asterisk. They take effect at the point they occur during the compilation of the Fortran source program. As with all other Fortran statements they may be contained with continuation lines, and only columns 7-72 are significant. All spaces, blank lines and comments within the directives are ignored, in a similar way.

10.5.1 End of Program

The *END directive is used to mark the end of the Fortran program, as an alternative to the EOT character.

10.6 INPUT/OUTPUT

Input/output is implemented to the Fortran Standard. In extension to this standard characters may be read or written from REAL, INTEGER or LOGICAL data types using the 'A' format (see also 10.2). When the LOGICAL type is used to store Holleriths, problems may arise. LOGICAL operations are performed as one bit operations delivering only a one bit result; consequently any previously existing Hollerith data may be lost.

The details of pre-connection are left undefined by the standard. In the implementaion unit numbers are related to MUSS streams for the purpose of pre-connection. The result of unit modulus 8 gives the MUSS stream number which is used for the transput. It is the first access to the unit which defines if it is an input or output stream. However, some operations on a pre-connected unit do not imply input or output (namely BACKSPACE, REWIND, INQUIRE, OPEN, CLOSE). In these cases there should not be an input and output stream of the same number, and then it is unabiguous which stream should be used.

The BACKSPACE statement cannot be used to move between sections of a multi-sectioned MUSS stream. REWIND operates differently, rewinding a unit causes the stream to be ended and re-connected on the next access. This affects output sent to a process; any output destined for a process will be sent when a rewind is made, each rewind causing a further document to be sent.

When an INQUIRE by name is made to a pre-connected file, the inquiry will not be able to determine which unit the file is connected to, unless the unit has been accessed previously in the same run. If such an inquiry is made only details about the file will be returned, and not any about the connection. An OPEN statement which gives no file name causes a connection to the MUSS current file. A PRINT statement or a READ with no unit number, or an '*' as a unit refers to input or output stream zero.

Output from programs consists of what was specified by the FORMAT, WRITE or PRINT statements. The carriage control characters in the first column are not interpreted or removed, this enables output to be read back by the same format that was used to write it (as the standard requires). To send any output to a printer the LIST command should be used. This has one parameter (the file to be printed), and it processes the file producing the correct interpretation of the carriage control characters for printing.

Fortran sequential input/output has a default maximum record length. In Standard Fortran there is no way of specifying the record length for sequential input/output so the subprogram FIO.SET.UNIT.REC.L is used. This subprogram has two parameters, the first the unit number and the second the maximum record length required. If this procedure is not used a suitable default is assumed. This procedure may be used for both program connected and pre-connected units. At present the ability to call library procedures from Fortran is not implemented, so FIO.SET.UNIT.REC.L is currently inaccessible.

10.7 IMPLEMENTATION DEPENDENCE

10.7.1 Accuracy

10.7.1.1 MU5

On MU5 INTEGER is in the range -2**31 to 2**31, REAL has a mantissa of 52 binary digits (approx 15 decimal digits) with an exponent range 16**1024 to 16**(-1024) which give a maximum decimal magnitude of 10**101233 approximately.

DOUBLE PRECISION has an accuracy of 31 decimal digits approximately.

10.7.1.2 MU6G

On MU6G INTEGER is in the range -2**31 to 2**31, REAL has a mantissa of 24 binary digits (approx 7 decimal digits) with an exponent range 16**64 to 16**(-64) which gives a maximum decimal magnitude of 10**77 approximately.

DOUBLE PRECISION has an accuracy of 16 decimal digits approximately.

10.7.2 Stop and Pause

For STOP and PAUSE statements with no arguments, a caption indicating that the program has stopped or paused together with source program line number of the STOP or PAUSE statement is output on stream zero. When STOP and PAUSE have arguments the string of digits or characters is output with the caption.

After obeying a STOP or PAUSE control exits from the program to the caller. A paused program may be re-entered by the command CONTINUE.

At present PAUSE will not return to the caller and CONTINUE is not implemented.

10.8 COMPILER PARAMETERS

The compiler parameters were previously mentioned in chapter 8.

The first parameter specifies the origin of the Fortran 77 source.

The second parameter specifies the destination of the compiler output.

The third compiler parameter specifies the mode of the compiler.

The fourth parameter specifies the maximum number of library interface procedures.

10.8.1 Mode Parameter

Bit 7 (the least significant bit) of the compilation mode is used to indicate that the interpretation of imbedded MUSS commands occuring during compilation is allowed.

Bits 8-15 of the compilation mode are used for debug monitoring. The significance of these bits is described in section 13 of the Fortran Compiler Implementation Manual.

CHAPTER 11 COBOL

Awaiting completion of the Cobol compiler.

CHAPTER 12. PASCAL

12.1 Introduction.

A Pascal compiler will be provided under MUSS. It is expected to conform to the proposed ISO standard when this is available.

CHAPTER 13 VIRTUAL STORE MANAGEMENT

The virtual store management procedures enable a process to manipulate its virtual store. The virtual store is assumed to be segmented, but the exact number of segments varies between MUSS installations. On MU6G, there are 128.

A process is initially created with only one segment (segment 0). A process may not change any of the properties of this segment or release it, as it is generally used as a stack and to hold the global variables, such as the PW's. A process is free to use other segments for the stack, and the organisational commands will operate using the curent stack area.

Further segments may be created by the user process, and subsequently the process can change the size or access permission of the segment, or release it to recover the space. There are five access permission bits associated with each segment, denoted AURWO.

A - Authorised to change. When this is set (=1) the process is allowed to increase its access permission to the segment.
U - User/executive bit. When set (=1) the segment is accessible by the process when in user mode.
R - Read permission bit. If set, the process may read the segment as data.
W - Write permission bit. If set, the process may alter the contents of the segment.
O - Obey permission bit. If set, the process may execute instructions from the segment.

When a segment is initially created, it is cleared to zero and has an access permission of %1E i.e. read and werite access in user mode, with authorisation to change the permission.

As the size and number of accessible segments is often too restrictive, particularly on small (16 bit) machines, a process is allowed a larger number of segments than can be addressed directly. Facilities are provided to allow a user to MAP a subset of the segments (the exact number is hardware dependent) into the addressable space. A special type of segment, known as X.SEGMENT, may also be created, whose maximum size is not determined by the physical characteristics of the central processor. Thus, for example, on a PDP11, X.SEGMENTS may be created which are larger than the virtual addressing space of a process (64K bytes). These segments may not be mapped and addressed directly, but may have individual blocks copied into/out of "normal" segments whenever they need to be accessed.

13.1 COMMANDS

(1) CREATE.SEGMENT (I,I)

This procedure creates a new segment on behalf of the current process. Parameter P1 specifies the segment number. If this is negative, then the system will choose a suitable segment number to allocate. The second parameter P2 is the segment size in bytes. This might be rounded up by the system to some suitable multiple of the page size. The maximum size of a segment is of course, restricted by the address translation hardware. On MU6G this size is 256K bytes. If the user requests a size greater than this, consecutive segments will be allocated to cover the required area. It should be noted, however, that in all subsequent operations, multiple segments are treated as separate and not as one contiguous area.

The create segment procedure returns two values. The segment number is returned in PW1. (This value is returned both when the system selects a segment number or when the number is specified in P1). If multiple consecutive segments are allocated, PW1 contains the number of the lowest segment assigned. PW2 returns the size of the area allocated in bytes. This may differ from P2 if rounding has taken place.

(2) RELEASE.SEGMENT (I)

This procedure removes the specified segment P1 from the users virtual store. If a file has been opened into this segment, the action of releasing it is in effect to close the file for the current process (and free other processes waiting for exclusive access).

(3) CHANGE.SIZE (I,I)

This procedure changes the size of the segment specified in P1 to the size in bytes given by P2. As with the create segment command the size may again be rounded to some suitable multiple of the page size. If the size is =<0, the effect is to release the segment. It should be noted that changing the size upwards to greater than the maximum segment size is invalid, and does <u>not</u> result in the creation of subsequent segments to allow for the upward expansion.

The size of the segment after this command is returned in PW1.

(4) CHANGE.ACCESS (I,I)

This procedure changes the access permission of the segment specified in P1. The access permission P2 is bit significant, with interpretation of the AURWO bits as explained earlier. If the A bit is currently set for the segment, the process is allowed to change its access to P2. If the A bit is not currently set, then only reductions in access permission are allowed, and parameter P2 is verified accordingly.

(5) INTERCHANGE (I,I)

This procedure interchanges the specification of the two segments given by the parameters. It is permissable for either (or even both) of the segments to be undefined, and in this case the effect is simply to redefine the existing segment.

(6) READ.SEGMENT.STATUS(I)

This procedure reads the status of the segment specified, and returns in PW1 and PW2 the size in bytes and access permission respectively. The access permission is in the form:

```
 ___________________________________
|_______|___|___|___|___|___|___|___|
          |   |   |   |   |   |   |
          |   |   |   |   |   |    ---Obey permission set
          |   |   |   |   |    ---Write permission set
          |   |   |   |    ---Read permission set
          |   |   |    ---User access allowed
          |   |    ---Authorised to change access
          |    ---X.segment
           ---Segment mapped
```

(7) CREATE.X.SEGMENT(I,I)

This procedure creates a single segment whose limit is not dependent on the segment size of the host machine. It may not be addressed directly, but should be used in conjunction with the COPY.BLOCK procedure to extract blocks of the segment and make them available in the virtual store of the process.

Parameter P1 specifies a segment number, as for CREATE.SEGMENT. P2 is the segment size in blocks (block size = 1024 bytes on PDP11).

The procedure returns two values. The segment number is returned in PW1, and the size allocated for the segment (in blocks) is returned in PW2. As for create segment, this may differ from P2 if rounding has taken place.

(8) CHANGE.X.SIZE(I,I)

The procedure changes the size of an x.segment. The segment number is specified in P1 and the size in blocks is given by P2.

The size of the segment after this command is returned in PW1.

(9) MAP(P1,P2)

On machines which have a limited number of addressable segments, this procedure may be called to make a segment P1 available for access at the position of segment P2 within the virtual store.

PW1 is set to the number of the segment which previously occupied that position in the virtual store.

(10) COPY.BLOCK(P1,P2,P3,P4)

This procedure copies information between two segments, and in particular, it should be used to copy information between x.segments and mapped segments wherever the x.segments have to be accessed. P1 and P2 specify a segment number and block number which is the source of the data, P3 and P4 specify the destination segment and block number.

CHAPTER 14 CREATION AND CONTROL OF PROCESSES

MUSS is designed as a system in which any process may create other processes and act as their supervisor. It is thus possible to extend the facilities of the basic system, by creating additional/alternative supervisor processes. In this Chapter, the facilities for creating and controlling processes are described.

14.1 PROCESS CREATION

Process creation is achieved by calling a command CREATE.PROCESS. Any process may use this command provided that it can supply the username and password of an authorised user of the system. The user will subsequently be charged for the resources used by the process. The creating process becomes the supervisor of the created process, and as such, it is allowed to use a special set of commands for controlling the process.

14.2 SCHEDULING

When initially created, a process is in a suspended state, and is not eligible for scheduling by the operating system. The supervisor may activate the process using the FREE.PROCESS command, and the process is then eligible for CPU time. The allocation of CPU time is based on the priority level associated with the process at the time of its creation. Sixteen levels of priority are recognised by the system, ranging from the highest priority 0, to the lowest 15. Charging for CPU time is related to the priority (on a non-linear scale), so that a higher priority job may be guaranteed an improved response time but at a considerably increased cost. User processes normally run at priorities in the range 8 - 15, and are assumed to have the following characteristics

15	-	background jobs (will be run sometime)
14	-	normal batch
13	-	high priority batch
12	-	express batch
11	-	low priority interactive
10	-	normal interactive
9	-	high priority interactive
8	-	express interactive and operator work.

Different scheduling rules apply to batch and interactive processes. A batch process, once started, is normally run to completion unless pre-empted by the arrival of a higher priority batch process. Interactive processes are timesliced, with the duration, cost and frequency of timeslices dependent upon the priority level.

14.3 CPU TIME

The maximum amount of CPU time to be made available to a process is specified at the time a process is created. A process can arrange to be interrupted after a specified amount of time has elapsed by calling SET.TIMER. This provides the means of subdividing the available time between various parts of a job. On being interrupted, the timer is automatically set to the maximum amount of time remaining. Note that a small amount of the total available time is normally withheld by the system for the purpose of monitoring in the event of using the requested amount of CPU time. The total amount of CPU time used may also be read using the READ.TIMER command.

14.4 INTERRUPTS

The interrupt for time expired is actually one of four types of interrupt trap which may be forced on a process. The reasons for interrupt traps fall in the categories

Trap 0	Hardware detected Fault conditions
Trap 1	Limit violations
Trap 2	Timer expired
Trap 3	Message interrupts or external interrupts (e.g. Pressing the break key on a terminal.

The action on the process of receiving one of these interrupts is to obey a forced procedure call to a designated trap procedure. The address of this procedure may be set using the command SET.INT.TRAP. A procedure READ.INT.TRAP also exists for reading the address of the trap procedure. The trap procedure must take steps to save and restore any registers which it might destroy, if it is expecting to return from the procedure and continue execution from the point at which the interrupt occurred.

14.5 COMMANDS

1) CREATE.PROCESS(II,II,II,I,I,P,I,I)

This procedure creates a process of name P1 on behalf of the user whose name and password are given in P2 and P3 respectively. If P1 is zero, a unique name will be generated for the process, and returned in PWW1. The process will run at the priority given by P7 and will commence execution at the procedure P6.

The maximum resources to be given to the process include a store limit P4 specified in K words (integers), and a CPU limit of P5 seconds.

The facility is also provided for the system to inform the supervisor when a process terminates. The parameter P8 defines a channel number on which the supervisor is prepared to accept a termination message containing the process identification. This channel number is encoded as for send message.

The identification of a process takes the form of a system process number (SPN) and a process identifier (PID). These are returned in PW1 and PW2 by the create process command, and must subsequently be used by the supervisor when calling the procedures for controlling the process.

2) TERMINATE.PROCESS(I)

This procedure is called whenever a process wishes to terminate. The parameter states a termination reason, which will be sent to the supervisor if a termination message was requested at create time. Negative fault reasons imply termination due to a fault condition. In particular, a reason of -100 implies that the process was aborted, and the system will attempt to salvage whatever monitoring information is available.

3) KILL(II,I)

This command may be called by a user to terminate the specified process. It will only operate if the process is running under the same user name as the current user (or if the current user is privileged). The parameter P2 states a reason for killing the process.

4) SUSPEND.PROCESS(I,I)

This procedure may be called by a process' supervisor to suspend execution of the specified process (P1 = SPN, P2 = PID).

5) FREE.PROCESS(I,I)

This command allows a process' supervisor to free it, and thus make it eligible for scheduling (P1 = SPN, P2 = PID).

6) LOOK.UP.PROCESS(II,II)

This procedure finds the internal identification of the process specified in P1. If P1 is zero, the identification of the current process is returned. The second parameter states the machine in which the process exists (0 implying the current machine). The identification takes the form of an SPN in PW1 and a PID in PW2.

7) READ.PROCESS.STATUS(I)

This procedure reads the status of the process with the specified SPN. If this is negative, the current process is assumed. The status information takes the form of

PWW1	Process name
PWW2	Username
PW1	SPN
PW2	Process status, which is bit significant with the following interpretation:

```
____________
|      | | |
         | |
         |  --suspended
          --waiting for message
            (or after sending message)
```

PW3	Priority
PW4	PID
PW5	Suspension reason (if any)
PW6	Start time.

8) CATALOGUE.PROCESSES CPR

This procedure returns a segment containing information about all the processes in the machine. The entries, which may be indexed by SPN, occupy 32 bytes each and have the following format:

Process name (8 bytes)
User name (8 bytes)
2 spare bytes
Status (1 byte)
Priority (1 byte)
Suspension reason
Process identifier PID
Start time

Each entry is 4 bytes unless stated otherwise. The total number of entries is returned in PW1, and PW2 holds the segment number.

9) SET.TIMER(I)

This procedure allows a process to set its local CPU timer. The parameter states the number of seconds, as in the create process command. To allow for cases where this may be greater than the total amount of CPU time remaining, the timer is set equal to the smaller of the two.

10) READ.TIMER()

This command returns in PW1 the amount of CPU time used by the current process. The maximum available time is also returned in PW2.

1) FORCE.INT(I,I,I)

This command may be called to force an external interrupt (interrupt trap3) on a process. P1 and P2 specify the SPN and PID of the process respectively, and P3 a reason to be passed to the trap procedure. The command will only operate if the current process is running under the same username or of the current user is privileged.

12) SET.INT.TRAP(I,P)

This command sets the address of the specified interrupt trap (P1 masked in the range 0 - 3) to be the procedure P2.

13) READ.INT.TRAP(I)

This command reads the procedure address for the specified trap and returns it in PW1.

CHAPTER 15 INTER-PROCESS COMMUNUCATIONS

Processes in MUSS communicate with one another by sending messages. Each process has eight input channels to which messages may be directed. Thus, the complete identification of the destination for a message is given by the system process number SPN and process identifier PID of a process and a channel number for that process. The message system is designed to be network wide, and so part of the SPN identifies the machine containing the destination process.

To enable a two-way communication to be set up easily, and to prevent a rogue process from injecting unidentified messages into the communication system, the system appends to each message the identification of the sender. This is returned to the destination process when it reads the message, and may subsequently be used as the address for reply messages. There is an additional problem in ensuring that messages sent as replies can be associated with a particular message from the original process. Processes can therefore specify a sequence number, which is presented to the destination process as part of the reply information.

A process also has control over which messages are accepted on a channel. Each of the eight channels can be set into one of three states: closed, open for all messages, and dedicated to one specific process. Any messages directed to a closed channel is rejected, any messages sent to an open channel is accepted and automatically queued. With having eight channels under its control, therefore, a process can selectively stream its input.

Each of the message channels acts as a first-in-first-out queue. There are a number of ways in which a process can tell whether there are messages waiting to be read. A process may poll its message channels by attempting to read messages from them. Alternatively, it may elect to wait until a message arrives on a specified channel, or one of a number of channels. In this case, a time limit is also specified, and the process is re-activated if no messages have arrived within that period. Finally, a process may request the status and condition of one of its message channels.

A message consists of two parts, a short header optionally accompanied by a segment from the senders virtual store. The header is copied from the sending to the receiving process' virtual store; the segment is unlinked from the senders virtual store, and linked into the receivers virtual store when the message is read. The format of data in a message is not fixed by the system, and is usually by arrangement between the sending and receiving processes. The system software and input/output controllers have certain conventions for messages, and these are described in Chapter 16.

15.1 COMMANDS

(1) SEND.MESSAGE ([C],[I],I,I,I,I)

This procedure sends a message to a specified channel of the specified process. P1 is the data to be sent in the short message. It should not be greater than 80 characters long, and exist in store normally accessible to the calling process. If longer messages are included, they are truncated after 80 characters.

The second parameter, P2, is a vector specifying the destination for the message. It should be at least four elements long, holding respectively

the SPN (and machine no) of the receiving process
its PID
the destination mode
the destination sequence number

The destination mode and sequence number are parameters included to help the setting up of "conversations" between two processes. They are encoded in the same way and have reciprocal functions to the parameters P3, the source mode and P4, the source sequence number.

The mode may take the following values:

	Source mode	Destination mode
%10	Suspend the process after sending the message	Free the destination process if this is a reply of the right sequence number.
%8-%FA	reply may be sent equal to the bottom 3 bits (i.e. channels 0 to 7)	The bottom 3 bits on a channel number specify the destination channel number on which to link the message.
%0	No reply expected	Ignore the send message request.

The sequence number, P4, provides a mechanism whereby a process can append an identifier on to an outgoing message. This identifier may then be used by the recipient process when making replies, so that the original process can associate outgoing and reply messages. The sequence number may take any user defined value. The only occasion when it is examined by the system is when a reply is sent to free a suspended process. In this case, the process must have suspended as a result of sending a message with a corresponding sequence number.

The parameters P5 and P6 represent a segment number and access permission respectively, for a segment within the process' virtual store. The segment must be present in the virtual store, and the same rules apply to changes in access permission as would apply to a process calling the change access command. If the segment parameter is less than or equal to zero, then no segment is sent.

(2) READ.MESSAGE ([C],[I],I,I)

This procedure reads a message from the channel specified by the bottom 3 bits of P3. P1 is a vector where the characters passed as the short message may be returned. It should exist in store normally accessible to the calling process. If the vector is not long enough to store the short message, the excess characters will be discarded and lost.

The parameter P2 should be a vector of at least 4 entries. Information about the source of the message is copied into this vector, in the form:

SPN of the source (including machine number)
PID of the source
Mode for replies
Sequence number for replies.

This, of course corresponds exactly with the destination parameter of the send message command. Thus, conversations may be set up by using this information to send reply messages. The only other information about the sender of the message, the identification of the user, is returned in PW4.

The fourth parameter specifies a segment number which may be used for long messages. If this is negative, a free segment number is allocated. PW1 always contains the segment number actually used (negative if a short message), PW2 contains the size of the segment in bytes and PW3 holds the access permission accorded.

(3) SET.CH.STATUS (I,I,I,I)

This procedure sets the status of the input channel specified in P1. The status, given in P2, is interpreted in the following way:

```
 ______________________________
|__________|_c_|_b_|_a_|
```

(a) Allow messages from any process.

(b) Dedicate the channel to allow messages from the process whose identifier (PID) is given in P3.

(c) Close the channel after 1 message corresponding to the specified sequence number P4. Reject all other messages.

A status of zero implies that the channel is closed to all incoming messages.

(4) WAIT (I,I)

This procedure halts the current process pending the arrival of a message. The first parameter defines the channels on which messages are expected. Thus, if the least significant bit is set, messages on channel 0 will wake the process; if the next bit, messages on channel 1 etc.

To avoid a process waiting an indefinite period of time for a message, a time limit in seconds, P2, may be specified. If a message on one of the required channels has not been received within this time, then the process is woken.

If the time limit is negative or zero, or a message is already on one of the required channels, then the procedure returns immediately without halting the process. PW1 always contains a bit pattern indicating the channels on which messages are present. Thus, the least significant is set if messages are on channel 0 etc.

(5) READ.CH.STATUS (I)

This procedure reads the status of the specified message channel. PW1 returns the status bits, as specified in SET.CH.STATUS, PW2 the PID if the channel is dedicated and PW3 the sequence number if it is dedicated for a specific message. The number of messages on the channel is returned in PW4.

CHAPTER 16 CONVENTIONS FOR NETWORK COMMUNICATION

In general, a process may send any information it wishes to another process, in a format mutually agreed by the sending and receiving processes. The system does not normally interpret the information given in either the short message or in any accompanying segment, unless the destination process is one of the system processes, such as a device control process. In this case, certain conventions are assumed for the format of messages. These conventions are described in this Chapter.

16.1 LAYOUT OF MESSAGES

Both short messages and segments containing data to be read by system processes, are assumed to contain a small amount of housekeeping information. This defines the size and location of the information, and also the type. Three basic types are defined, namely:

characters
binary information
records.

The distinction is also made between fixed and variable length records.

The housekeeping information takes the form of a preamble at the start of the segment or message. This occupies 12 bytes in the segment and 6 bytes in a short message.

The first 32 bits in a segment (or first byte in a short message) contain an integer which is the byte displacement to the start of the useful data from the beginning of the segment/message. The second 32 bits (or byte in a short message) contain an integer count of the number of bytes within the data in the case of character or binary documents, or the number of records. The size of the preamble is not included in this count.

The remaining four bytes in the preamble have functions dependent on the type of document. Byte 3 is used to indicate the type:

%00	Characters in segment/message
%01	Binary
%02	Unit (fixed length records)
%03	Variable length records.

In the case of fixed length records, bytes 0 and 1 hold the size of the records. The use of these bytes is not defined for the other types of documents.

The only other housekeeping information assumed by the system is a two byte field preceding every record in the variable length case. This gives the size of the record which follows.

16.2 COMMUNICATION WITH PERIPHERAL CONTROLLERS

All connections into or out of a machine go via one of the system peripheral control processes. These decipher the information as it passes through, in order to decode directives to the peripheral management system. Two types of peripheral connection are possible, namely that with a character oriented device and a communications line. These have slightly different conventions, owing to their different treatment of messages. They are, therefore, described separately below.

Input via a character oriented device.

A process controlling a device such as a terminal or paper tape reader, has conventions for denoting the start and end of documents and for distinguishing between input via short and long messages. In general, a user must log a device on to a specific process. A sequence of short messages or a single long message may then be transmitted to that process.

In the short message case, the device is logged in by placing the command

```
***M processname
```

at the start of a line. The device is logged in to the named process and the remainder of the line after the process name is passed as the first message to the process. Each subsequent line is passed as a separate short message to the same process, until either a further logging in command or a logging out (***Z) command is detected at the start of a line.

Long messages may be sent by using the logging in sequence

```
***A processname
```

instead of ***M. The remainder of the line after the process name is used as the header for the long message. All subsequent input data up to the terminator (***Z) is placed in a segment, using the character document conventions described earlier. The message is then dispatched to the specified process and the device logged out. Only ***Z may act as a terminator, and any other sequence of *** at the start of a line will be faulted.

Output via a character oriented device.

Each output device has associated with it a particular message channel, from which it can accept and process messages. The messages must conform to the character document conventions described earlier. In the case of a device which can accept messages from more than one process at a time, such as a lineprinter control process, the channel is normally set open to all processes and may not be logged on to a specific process.

For more dedicated peripherals, such as a teletype or VDU, the device must be logged on to one specific process and the message channel will be dedicated accordingly. This may be achieved by sending

***M processname

to the message channel, where the process name specified is the one to be logged in. Quite obviously, if the device is currently logged in, the new log in request can only come from the currently logged in process. A device may similarly be logged out with a ***Z message.

Many output devices are regarded as being paired with an input device, such as the keyboard and screen of a VDU. In this case, both "devices" will be logged in to the same process, regardless of whether the command was received on the input device or the output message channel.

Communication Lines

Data transmitted via communications lines carries additional control information which distinguishes between the long and short message forms. This information is inserted by the device driving mechanism, and so only a single logging in sequence is necessary. Thus logging in and logging out is performed by ***M and ***Z appearing at the start of a short message. Apart from the examination of the start of short messages, the data transmitted is not inspected or interpreted by the communications software.

In addition to the ***M and ***Z conventions, the system software will also take account of four consecutive asterisks (****) appearing at the start of a short message. This is to allow logging in messages to be transmitted via communication lines from a process in one machine to a peripheral control process in another. The four asterisks are replaced by three, and the data is then transmitted without being interpreted further within the source machine.

It is anticipated that further *** commands will be introduced to allow the transmission of configuration information for both the character and communication peripherals. To be defined later.

CHAPTER 17 DISC AND MAGNETIC TAPE FACILITIES

This Chapter outlines the procedures provided in MUSS for handling exchangeable disc and magnetic tape devices. The facilities described are used by system software such as the file archive manager, but are also available to users wishing to access private volumes outside the file system.

The basic system distinguishes between two types of device: fixed and variable block transfer devices respectively. Most exchangeable discs and some magnetic tapes are fixed block devices. These are pre-addressed, and hence a device address can be specified with each transfer request. Industry Compatible Magnetic Tapes, as well as some exchangeable disc systems, are variable block devices. With these there is no concept of 'address' on the device and accessing is usually sequential. For convenience, fixed block devices are referred to as 'discs', and variable block devices as 'tapes'.

Individual tapes and disc cartridges (volumes) are identified to the system by a name and a label. The name is used to identify the volume to the operators - thus it will usually be a serial number enabling the operator to find the required volume upon request. The label is under the control of the user who 'owns' a particular volume, and serves two purposes. Firstly, it provides a check that the correct volume, and not another which has somehow acquired the same serial number, is loaded. Secondly, it acts as a 'password', protecting the volume against unauthorised access by other users. Labels are written by one of the procedures described below. A facility exists to force the system to accept an incorrectly labelled volume. This enables non-standard volumes, and volumes which have not yet been labelled, to be accessed. This facility is described with the operator commands in Chapter 10.

In the basic system, deadlock situations are resolved by the operator by dismounting selected volumes. Thus, all processes using discs or magnetic tapes should be prepared to receive the response 'device not available' to any of its requests, and to recover as appropriate.

17.1 COMMAND PROCEDURES

The commands associated with disc and tape management fall into two categories, namely the acquisition and control of a disc/tape unit and the commands for actually driving the unit, performing transfers etc. The high level control functions are common to both tapes and discs, and a description of these commands immediately follows. The physical characteristics of the two types of media require a different set of functions for performing transfers, and so these commands are described separately for the fixed and variable block devices.

17.1.1 Organisational Functions

1) MOUNT(II,II,S)

This procedure requests a particular tape or disc to be mounted on an available unit, and allocated to the calling process. The unit number is returned in PW1. The first parameter is used to signify whether the medium is a tape or disc, and so P1 should take the value "TAPE" or "DISC". P2 is the name of the tape/disc to be loaded. If it is not already available on a drive, then a message is output to the operator to mount it. This message will be repeated periodically until either the tape/disc becomes available or until the operator signals CANTDO to the process.

The expected value of the tape label appears in P3. If a tape of the required name is available but P3 does not match the label on the tape, an error is signalled to the process.

2) RELEASE(I)

This procedure is used to release a disc or tape unit previously allocated by the MOUNT command. The device is disengaged and the operator informed so that the volume may be removed from the drive. P1 specifies the unit number to be relinquished. If this is negative, all disc and tape drives assigned to the current process will be released.

3) WRITE.LABEL(I,II,[C])

This procedure changes the name of a volume which has already been mounted. P1 specifies the unit number, as returned from the mount command. P2 and P3 give the new name and label to be written to the volume.

17.1.2 Fixed Block Transfers

1) ED.READ(I,I,I,I)

This procedure reads data from a disc or pre-addressed fixed block size magnetic tape to a buffer in the current process' virtual store. P1 states the unit number of the required volume. P2 gives the virtual address of the start of the buffer. It is expected that this will start on a block boundary (where the block size is hardware dependent), and so the bottom address bits will be ignored to ensure this. P3 gives the starting block number on the disc and P4 the number of blocks to be transferred.

NB. On MU6G, the block size is 2Kbytes.

2) ED.WRITE(I,I,I,I)

This procedure writes data to a disc or pre-addressed magnetic tape from a buffer in the current process' virtual store. The interpretation of the parameters is exactly the same as for ED.READ.

17.1.3 Variable Block Transfers

1) MT.SKIP(I,I)

This procedure skips over a block of data on the specified device. P1 is the unit number and P2 the direction to be travelled (0 = forwards, 1 = backwards). PW0 might be set to a fault condition if the device tries to skip over a tape mark, or beyond the end of the tape.

2) MT.READ(I,I,II)

This procedure reads the next block of data from the specified unit (P1). P2 gives the virtual address in bytes to which the data is to be read and P3 states the maximum size of the transfer. P4 gives the direction in which the medium is to be moved (0 = forwards, 1 = backwards). As with MT.SKIP, a fault will be signalled if a tape mark or end of tape is encountered.

The actual size of the transfer (in bytes) is returned in PW1.

3) MT.WRITE(I,I,I)

This procedure writes a block of data to the specified unit (P1). P2 gives the virtual address in bytes of the start of the data and P3 the number of bytes to be transferred.

4) MT.SKIP.TM(I,I)

This procedure skips across data blocks on a medium until the next tape mark is encountered. P1 specifies the unit number of the tape and P2 the direction of travel (0 = forwards, 1 = backwards).

5) MT.WRITE.TM(I)

This procedure writes a tape mark on the specified device (P1).

6) MT.REWIND(I)

The procedure rewinds the specified tape so that the transfer heads are positioned at the start of the medium.

17.2 UTILITIES

A number of facilities are provided to allow users to read and write tapes according to certain standard formats.

The following procedures read/write tapes which conform to the ANSI standard X3.27, regarding the format and use of labels and tape marks.

1) WRITE.FILE(I,[C],[C],[C],[C],I)

This procedure writes a file P2 to tape unit P1. The file is written in 1K byte blocks, preceeded by an ANSI reader label, and followed by an end of file label and tape mark.

Parameters P2 and P3 allow for the insertion into the labels of a sequence number of the file on the tape, and a file generation number respectively. P4 is an expiry date for the file. This is specified as 5 digits, the first two giving the year, and the next three the day within the year. P6 is an accessibility key for the file. Suitable defaults for P2 to P6 are supplied if specified values are not specified.

2) WRITE.TAPE(II,[C])

This procedure ma be caled to write a set of files to a tape. In addition to the file labels, suitable volume labels are tape marks are written to the tape, according to ANSI standard.

The parameters P1 and P2 specify a tape name and label respectively. The procedure will prompt for the names of the files to be written to the tape. To complete the tape and exit from the procedure, a filename of @ should be given.

3) READ.FILE(I,[C],[C],I)

This procedure reads a file from tape, assuming it is in the correct format. P1 specifies the unit number of the tape drive. P2 gives the filename which will be used for securing the file. If this is null, the filename wi;ll be extracted from the file header label on the tape. P3 should be a vector into which the 80 character header label is copied. P4 gives the size of the data blocks on the tape.

4) READ.TAPE(II,[C],I)

This procedure reads a set of files from a tape. The tape name and label are specified in P1 and P2 respectively. The names of the files are extracted from the header labels preceding each file on the tape. P3 specifies the size of the data blocks. If zero, a block size of 1K bytes is assumed.

5) COPY.TAPE(II,[C],II,[C],I)

This procedure may be called to create a copy of a tape. P1 and P2 give the name and label of the tape to be copied. P3 and P4 give the name and label of the tape to be overwritten. The parameter P5 gives the action to be taken on errors. If non-zero, the copying process is halted on detection of an error. When zero, errors are monitored, biut an attempt is made to continue the copying sequence.

CHAPTER 18 BENCHMARKING AND PERFORMANCE MEASUREMENT

Two procedures are provided for noting and printing the statistics collected about the system:

1) READ.STATS()

This procedure notes the current values of the system statistics.

2) OUT.STATS(I)

This procedure prints the statistics collected. Two options are available. If the parameter is zero, the difference in statistics since the last call of READ.STATS will be printed. Thus a program can monitor the performance of the system between any two points.

If the parameter is non-zero, or if READ.STATS has not been called, the statistics printed relate to the behaviour of the system since restarting.

STATISTICS COLLECTED

Address	
0	Number of organisational commands called
2	Ticks at user level (1/50 / 1/60 sec)
4	Ticks at command level
6	Ticks at interrupt level
8	Number of drum transfers in
A	Number of drum transfers out
C	Total number of blocks transferred
E	Elapsed time (seconds)
10	Number of processes created
12	Number of segments created
14	Number of X segments created
16	Number of long messages sent
18	Number of short messages sent
1A	Number of files opened
1C	Number of files created
1E	Number of files updated
20	Total number of tasks set.

CHAPTER 19 ACCOUNTING AND USER MANAGEMENT

Controlling users and accounting for their use of the system is dependent in conditions in a particular installation. The aim of the MUSS accounting system is to provide a set of primitive operations that the System Manager of an installation can use to build an accounting regime appropriate to his needs.

The basic system maintains accounts for five resources. These are:

1. OUTPUT	The number of Kbytes of data sent to output devices.
2. CPUTIME	The amount of CPU time used.
3. FILESTORE	How many Kbytes of filestore the user can occupy.
4. NO.FILES	The maximum number of files that the user can have in his directory at one time.
5. SUBORDS	The maximum number of subordinates the user can have.

Some high-level procedures are provided along with the basic system to be used as an example of the use of the primitives and to provide simple facilities for those who need no more.

19.1 COMMANDS

The primitive level commands and the high-level procedures are listed below. User SYSTEM effectively has infinite amounts of all resources which it can distribute to its subordinates who are all the other users in the installation.

19.1.1 Primitive Commands

The set of balances, incomes and maxima for a user is called his "parameters". The parameter numbers for the standard resources are:

OUTPUT	-	0
CPUTIME	-	1
FILESTORE	-	2
NO.FILES	-	3

SUBORDS - 4

1) CREATE.USER(II)

This command creates a new user with the name specified in P1 as a subordinate of the current user. The name must be unique within the installation.

2) ERASE.USER(II)

Removes the subordinate user with name P1 from the system. The subordinate's balances of consumables and maxima of re-usables are added to the current user's corresponding values. The subordinate must not have any subordinates.

3) SET.USER.PARAM(II,I,I)

This procedure re-distributes a resource balance, maximum or income between a user and one of his subordinates. P1 contains the name of the user, P2 is the parameter number. P3 is the amount to which the parameter is to be set. The change must not cause either user to overdraw his balance, or exceed his income or maximum.

4) NEW.PASSWORD(II,II)

Changes the password of the current user (P2 zero) or one of his subordinates (P2 is username) to the value in P1.

5) CATALOGUE.USERS(I)

Returns a new segment containing a table of information on the current user and his subordinates (if any). There is one entry for each subordinate and a subordinate's subordinates immediately follow his entry (defined recursively). The format of each entry in the table is:

```
 _________________________
| USERNAME (8 bytes)      |
| PASSWORD (8 bytes)      |
|      1 spare byte       |
| LEVEL (1 byte) *        |
| BALANCE 1 (2 bytes)     |
|                         |


|_________________________|
| BALANCE m (2 bytes)     |
| INCOME 1 (2 bytes)      |
|                         |


|_________________________|
| INCOME m (2 bytes)      |
| MAXIMUM 1 (2 bytes)     |
|                         |


| MAXIMUM n (2 bytes)     |
```

* This will be 0 for current user, 1 for immediate, subordinate, 2 for subordinate's subordinate, and so on.

PW1 holds the number of users in the table, PW2 holds the segment number. PW3 is set to the maximum number of subordinates that the user is allowed to create. PW4 holds the number of consumable resources, PW5 the number of re-usable resources (the m and n values respectively in the figure above).

6) PAYOUT()

This procedure causes each user's income for each consumable resource to be added to his balance and then, if necessary, this reduced to his limit. It may only be invoked by a trusted system user.

19.2 HIGH-LEVEL COMMANDS

1) NEW.USER([C])

Creates a new user as a subordinate of the current one and transfers to him a share of the current user's resources. How big a share is determined by the current user inputing positive integers each of which specify how much of a resource income or maximum is to be transferred.

When reading from the current stream, NEW.USER will prompt with the names of resources to which the creating user has access. The prompts will be issued in the same order every time the command is used. If no value is specified in response to a prompt, the subordinate is not given access to that resource. A slash ("/") will terminate reading of values and set any ones not yet prompted for to "no access".

Optionally, P1 can specify a stream from which these values are to be read. When the end of the stream is detected (by reading a slash or end of file character), NEW.USER will return to the currently selected stream for one or more sets of usernames and passwords.

The procedure will terminate with an error message if the user tries to create more subordinates than he is allowed.

2) DELETE.USER([C])

Removes a subordinate from the system, transferring his resource allocations back to his superior. The subordinate must not have any subordinates. The subordinate's files will be transferred into the current user's directory, or deleted depending on the setting of a "file action parameter". P1 is the name of a stream (the default is the current stream) from which usernames and file action parameters will be read. The file action parameter is either "T", meaning transfer the files of "D" indicating their deletion.

3) SET.ACCOUNT([C])

This command redistributes balances, incomes and maxima between a user and one or more of his subordinates. It works in a similar way to NEW.USER by prompting for changes to balance and income for each consumable resource, and maximum for each re-usable resource.

P1 is the name of a stream from which positive integers indicating the new values of the user's parameters are to be read. The currently selected stream is the default.

Usernames will be read from the currently selected stream after the new values, until a slash ("/") or end of file character is read.

4) LIST.ACCOUNTS(C,[C])

This command lists the accounts parameters for the current user (P1 equal 0), and his immediate subordinates (P1 equals S), or all his subordinates (P1 equals A).

For each user, his name and the amount used and amount left to use of consumables, and maximum amount of re-usables under suitable headings. P2 is the destination stream name (the currently selected is the default).

CHAPTER 20. OPERATOR FACILITIES.

1) DRIVE.PERI(I,I)

This procedure may be called by an operator to drive a device on a specific multiplexor channel. The channel number is specified in P1. The procedure processes messages on input stream 0 of the process, and produces output on the device according to the mode given by P2.

The mode is encoded in the following way:

```
 _____________________
 I_____I__I__I__I__I__I__I
        |  |  |  |  |  |
        |  |  |  |  |   --BREAK interrupt required
        |  |  |  |  |       after each section.
        |  |  |  |  |
        |  |  |  |   ----BREAK interrupt required
        |  |  |  |          after each newpage.
        |  |  |  |
        |  |  |   ------Transmit XOFF after each
        |  |  |            newline character.
        |  |  |
        |  |   --------Transmit ETX after each
        |  |               newline character.
        |  |
        |   ----------Remove carriage returns.
        |
         ------------Do not print headers.
```

N.B: The modes for certain common types of devices is given below:

Benson plotter	%35
Diablo printer (as a document printer)	%2F
Diablo printer (as a lineprinter)	%C

2) RELABEL(I,II,[C])

This command allows an operator to associate a new name and label with the tape on a tape drive. P1 specifies the tape unit. P2 and P3 give the name and label of the tape respectively. Subsequent requests to mount a tape (see Chapter 17) will recognise the tape by this new name and label, rather than by any label on the tape.

This command should be called prior to mounting the tape on the unit.

3) CANT.DO(II)

This command may be called by the operator to refuse the request by a process for resources (such as the mounting of a tape or exchangeable disc). The parameter gives the name of the process.

4) SET.TIME.AND.DATE(II,II)

This command should be called by the operator immediately on starting up the system. The first parameter is the time of the day, consisting of the six characters for HRS HRS MINS MINS SECS SECS. The second parameter is the date, again expressed by six characters, DAY DAY MONTH MONTH YEAR YEAR.

The operator should check the validity of the time and date after calling this command, using the procedures OUT.TIME and OUT.DATE (Chapter 4).

CHAPTER 21 LIBRARIES

It is clear from the description of MUSS that its operation is very dependant on the availability, to every virtual machine, of a substantial collection of procedures in executable binary form. They are called the system library. Facilities are also provided for the user to augment the system library with private libraries of executable code. This Chapter is concerned with the structure of these libraries and with the commands that allow compilers to compile calls of library procedures and command language processors to implement interpretive calls.

21.1 LIBRARY SEGMENTS

The compiling system (MUTL) generates library segments. These may be loadable or executable binary. We are concerned only with the latter and they can always be obtained by loading the former. It may be relevant to note that binding of calls on other library procedures occurs when executable binary is created.

A library segment contains the following

a list of names for the procedures it contains
their parameter specifications
a procedure entry table
the procedure bodies.

The detail structure is machine dependent and is only relevant to the procedures described in this Chapter and to MUTL. However, the following properties are standard.

1) The segment can be recompiled and, providing it contains the same procedures in the same order (in fact it is the order the procedure specifications rather than the order of the procedures themselves) with unchanged specification, previously bound calls will continue to operate.

2) If it is recompiled as in (1) new procedures may be added.

21.2 THE SYSTEM LIBRARY

At the time a MUSS system is generated, a selection of library segments can be included as the system library. The FINDN routine described below provides the access to this system library hence it must 'know' which segments are included. This knowledge takes different forms in different implementations. For example, on machines with large virtual memories their virtual segment numbers may suffice. On other machines it might be better to make another copy of the name (and property information) to make it more easily accessed by FINDN.

21.3 PRIVATE LIBRARIES

The command LIBRARY introduced in Chapter 2 of the User Manual, dynamically extends the set of procedures available through FINDN. Obviously its implementation is closely linked with that of FINDN.

21.4 PROCEDURES FOR LIBRARY USE

1) FINDN([C])I

This procedure searches the lists of names associated with both private and system libraries. It returns an integer (called a 'library index') that can be used as a parameter of other procedures. The value 0 means the name is not present in the library. Normally the 'library index' is only used as a parameter to other procedures such as FINDP and the TL.PL, and users do not need to know the encoding. In fact the encoding is machine dependent. On large virtual store machines the 'library index' will be the address allocated to the procedure in the library entry table together with a bit to indicate whether it is an organisational command or an ordinary non-privileged library procedure. On the small virtual store machines it will contain

```
          the number of the segment containing the procedure
          the address in the entry table for the 'MAPPED' segment
          two kind bits to distinguish
              permanently mapped library procs
              not permanently mapped library procs
              organisational commands.
```

2) FINDP(I,I)I

This procedure is given a library index as P1 and an integer as P2. It returns the type of parameter P2 of procedure P1 in MUTL encoding. If P2 = 0 it returns the number of parameters and if P2 is this number + 1 it returns the result type of the procedure.

3) PARAMETERS(I)I

This procedure returns the number of parameters associated with procedure P1.

21.5 INTERPRETIVE CALLING OF LIBRARY PROCEDURES

Sometimes it is necessary to interpret user source as immediate calls on library procedures. This occurs for example, in the PCS INTERPRETER. Thus it is necessary to stack a link, stack each parameter and then enter the procedure. These operations are at too low a level to be performed by the library procedures, therefore built-in functions are provided in MUSL.

CHAPTER 22 MISCELLANEOUS COMMANDS

1) TIME.AND.DATE()

This procedure returns in PWW1 the current time and date, expressed as the number of seconds since midnight on 1st January, 1900.

APPENDIX 1 INDEX OF COMMANDS

Command	Abbreviation	Chapters
BACKUP.FILE		5
BREAK.INPUT	BI	4
BREAK.OUTPUT	BO	4
CANT.DO		20
CAPTION	CAP	4
CATALOGUE.FILES		5
CATALOGUE.PROCESSES		14
CATALOGUE.USERS		19
CHANGE.ACCESS		13
CHANGE.DEST		3
CHANGE.SIZE		13
CHANGE.X.SIZE		13
COBOL		8, 11
COPY.BLOCK		13
COPY.FILE	CF	2
COPY.TAPE		17
CREATE.PROCESS		14
CREATE.SEGMENT		13
CREATE.USER		19
CREATE.X.SEGMENT		13
CURRENT.INPUT		4
CURRENT.OUTPUT		4
DEFINE.INPUT	DI	3
DEFINE.IO	DIO	3
DEFINE.OUTPUT	DO	3
DEFINE.STRING.INPUT	DSI	3
DEFINE.STRING.OUTPUT	DSO	3
DELETE	DEL	2
DELETE.FILE		5
DELETE.LIB		2, 8
DELETE.PROG		8
DELETE.USER		19
DRAW		6
DRIVE.PERI		20
ECHO.LINE		4
EDIT	ED	2, 5
ED.READ		17
ED.WRITE		17
END.INPUT	EI	3
END.OUTPUT	EO	3
ENTER.TRAP		7
ERASE.USER		19

Command	Abbreviation	Chapter
MAP		13
MOUNT		17
MT.READ		17
MT.REWIND		17
MT.SKIP		17
MT.SKIP.TM		17
MT.WRITE		17
MT.WRITE.TM		17
MUSL		8, 9
NEW		2
NEWLINES	NL	4
NEW.PASSWORD		19
NEW.USER		19
NEXT.CH		4
O.BPOS		4
OLD		2
O.MODE		4
OPEN.DIR		5
OPEN.FILE		5
O.POS		5
O.SEG		4
OUT.BACKSPACE		4
OUT.BIN.B		4
OUT.BIN.S		4
OUT.CH		4
OUT.DATE		4
OUT.FN		4
OUT.HRD		4
OUT.HEX		4
OUT.I		4
OUT.LINE		4
OUT.LINE.NO		4
OUT.M		7
OUT.NAME		4
OUT.OCT		4
OUT.REC		4
OUT.STACK		4
OUT.STATS		18
OUT.T		7
OUT.TD		4
OUT.TIME		4
OUT.UNIT		4
O.VEC		4

Command	Abbreviation	Chapter
PASCAL		8, 12
PAYOUT		19
PERMIT		5
PPC.CMD		2
PPC.SEQ		2
PROMPT		4
RD.DEBUG		7
RD.DUMP		7
RD.TRAP		7
READ.CH.STATUS		15
READ.FILE		17
READ.FILE.STATUS		5
READ.INT.TRAP		14
READ.MESSAGE		15
READ.PROCESS.STATUS		14
READ.RECOVERY.STATUS		7
READ.SEGMENT.STATUS		13
READ.TAPE		17
READ.TIME		14
READ.TRAP		7
RELABEL		20
RELEASE		17
RELEASE.SEGMENT		13
REMOTE.DIR		5
RENAME	REN	2
RENAME.FILE		5
RUN		2
RUN.JOB		2

Command	Abbreviation	Chapter
SAVE		2
SECURE.FILE		5
SELECT.DOC		4
SELECT.HEADER		4
SELECT.INPUT	SI	4
SELECT.OUTPUT	SO	4
SEND.MESSAGE		15
SET.ACCOUNT		19
SET.CH.STATUS		15
SET.I.BPOS		4
SET.INT.TRAP		14
SET.O.BPOS		4
SET.RECOVERY.STATUS		7
SET.TIME.AND.DATE		20
SET.TIMER		14
SET.TRAP		7
SET.USER.PARAM		19
SKIP.LINE		4
SPACES	SP	4
SPELL		6
STOP	STP	2
SUSPEND.PROCESS		14
TERMINATE.PROCESS		14
TEXT		6
TIME.AND.DATE		22
WAIT		15
WRITE.FILE		17
WRITE.LABEL		17
WRITE.TAPE		17

APPENDIX 2 CHARACTER CODES

A standard character set, based on the ISO representation, is used on all the machines in the MU6 complex.

Hex Code		Hex Code		Hex Code		Hex Code	
00	Null	20	space	40	@	60	
01	SOH (start of heading)	21	!	41	A	61	a
02	STX (start of text)	22	"	42	B	62	b
03	ETX (end of text)	23	#	43	C	63	c
04	EOT (end of transmission)	24	$	44	D	64	d
05	ENQ (enquiry)	25	%	45	E	65	e
06	ACK (acknowledge)	26	&	46	F	66	f
07	BEL (bell)	27	'	47	G	67	g
08	BS (backspace)	28	(	48	H	68	h
09	HT (horizontal tab)	29	)	49	I	69	i
0A	LF (line feed)	2A	*	4A	J	6A	j
0B	VT (vertical tab)	2B	+	4B	K	6B	k
0C	FF (form feed)	2C	,	4C	L	6C	l
0D	CR (carriage return)	2D	-	4D	M	6D	m
0E	SO (shift out)	2E	.	4E	N	6E	n
0F	SI (shift in)	2F	/	4F	O	6F	o
10	PLE (data link escape)	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	STOP	34	4	54	T	74	t
15	NAK (negative acknowledge)	35	5	55	U	75	u
16	SYN (synchronous idle)	36	6	56	V	76	v
17	ETB (end of transmission block)	37	7	57	W	77	w
18	CLCL (cancel)	38	8	58	X	78	x
19	EM (end of medium)	39	9	59	Y	79	y
1A	SUB (substitute)	3A	:	5A	Z	7A	z
1B	ESC (escape)	3B	;	5B	[	7B	{
1C	FS (field seperator)	3C	<	5C	\	7C	¦
1D	GS (group seperator)	3D	=	5D	]	7D	}
1E	RS (record seperator)	3E	>	5E	^	7E	~
1F	US (unit seperator)	3F	?	5F	-	7F	delete

APPENDIX 3. FAULT REASONS

Class 0 - Program errors
Type

Class 1 - Limit violations

Class 2 - Timer runout

Class 3 - External interupts

Class 4 - Organisational command errors

Type	
1	Segment already exists
2	No segments available in virtual store
3	Illegal segment number
4	Illegal size requested
5	Illegal access permission requested
6	Segment does not exist
7	System unable to create segment (no resources)
11	Illegal channel number
13	No messages on specified channel
14	System unable to send message (no resources)
15	Channel closed
20	System process number or process idenifier invalid
21	Current process not supervisor
23	System unable to create a process (no resources)
25	Illegal username or password
32	Process name aleady exists
33	Process does not exist
41	File store limit exceeded
43	File name does not exist
44	Exclusive access deadlock
45	Unauthorised access requested
50	Unable to update common segment
70	Mount abandoned
71	Label incorrect
72	Buffer non-existent or too small
73	Illegal unit number
74	EOT/TM encountered
75	MT operations failed
76	Unit not assigned to process

Class 4 - errors are initially returned
to a process by a negative type
code in PWO

Class 5 - Input/Output set up errors

Type	
1	No streams available
2	Current File not available
3	Illegal stream name (in STRn*)

4 Stream undefined
5 Incorrect type of stream
6 Incorrect type of document
7 Unable to open File

Class 6 - Input/Output errors

Type
1 Input ended
2 Attempt to read/write beyond end of unit or record
3 Output limit exceeded
4 Unable to create segment for buffered output document
5 Unable to file/send output document
6 Destination process does not exist
8 Illegal symbol in number
9 Illegal delimiter

Class 7 - JCL command errors

Type
1 Command line not recognised
2 Command name unknown
3 Illegal type of parameter
4 Unable to change command stream (via IN command)

Class 8 - Programming language run time errors

Class 9 - Basic applications errors