MUSS

USER MANUAL ISSUE 10

Printed by

Research Department and

Office Information Systems

Blue Bell, Pa

18 November 1982

SPERRY UNIVAC
COMPUTER SYSTEMS

COMA

Contents

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 as required, giving each process the impression that it has a complete machine to itself.

1.2. VIRTUAL MACHINES

The kernel of the MUSS operating system provides each process with a "virtual machine" in which to execute. The virtual machine consists of

(i) A share of the available processor time. As far as the process is concerned, it appears to have a complete processor to itself and need not be concerned with the fact that it is shared with other processes.

(ii) A share of the available store. In MUSS each process has a large virtual store which is divided up into logical units of information called segments, each of which can be up to 256K bytes long. Some of the segments are private to the process, and cannot be accessed at all by other processes. Others are common to all processes, and contain all of the system software; generally, processes are prevented from modifying the information in these common segments.

(iii) A library of procedures which the process may call. These reside in the common segments and so are available to all processes. They can be thought of as built-in operations, extending the instruction set provided by the actual processor.

An important feature of the virtual machine is that each process appears to have an entire machine to itself, with complete freedom to organise itself within this machine. Thus, processes need not be written in the knowledge that they will be sharing the computer with other processes. Input/output is dealt with by calling library procedures, which give each process the appearance of having its own set of input/output devices.

1.3. MESSAGES

The processes of an operating system are not in general independent of one another, but need to communicate. In MUSS, processes communicate with one another by sending and receiving messages (using built-in library procedures to do this).

Every process in the system has a unique name, by which the other processes (and users) may refer to it, and any process can send a message to any other by specifying its name. (In fact, an internal process identification consisting of a System Process Number (SPN) and a Process Identifier (PID) is used, but these can be found from the process name). A single message may contain any amount of information from a few bytes to 256K bytes.

The message system also forms the only means of communication 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 MUSS (except as stated below) is a set of compiled (executable) 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 remaining chapters in this Manual are mainly concerned with specification and use of these procedures.

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.

There is symmetry in the system between user programs and users typing commands on terminals. Either may use any library procedure. Also the user can extend the set of procedures available to him by introducing his own private libraries as described later.

Some implementations of MUSS on small machines may not have all of the above facilities in the system library. In this case the omitted facilities are normally available as private libraries (Chapter 21) under the username UTIL. To aid in recognition, these libraries usually have the name of the principal procedure which they contain. If any unexpected error 7/2 "Command Name Unknown" is received, users are advised to check if the required command is implemented as private library under UTIL. Users are warned that there may be local additions under this username which may not be documented in this Manual.

1.6. ACCOUNTING

The accounting system on MUSS is designed to ensure that all users get a fair share of the facilities available. Each object in the system that can be consumed or borrowed from a central pool is called a resource. An example of a resource which is consumed is line–printer output. Space in the filestore is a reusable resource; it is borrowed from the system when a new file is created and returned when the file is deleted.

For each consumable resource that he is allowed to use a user has a balance which records how much more he can use before he becomes overdrawn. The units which indicate the balance are a kind of money used to control the allocation of system resources. Within each resource, charges may be made at different rates depending on the quality of service provided. Thus one second of CPU time at Priority 8 will deplete a user's balance more than one second at Priority 9.

Similarly, for each reusable resource, there is a value indicating the maximum number of objects of that type that he can borrow at any instant (the maximum number of files in his directory for example).

At regular intervals (usually every day) users will have a fixed amount added to their balances of consumable resources. These "regular intervals" are termed accounting periods and the fixed amount, an income. If a user doesn't consume all of his income in an accounting period, then his credit in that resource will accumulate up to a limit set by the System Manager.

One resource a user may be given by the System Manager is that of having subordinates: that is, other users to whom he gives a portion of his allocation of resources. He becomes, in effect, a "local System Manager" creating and deleting users and controlling their use of resources.

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 XON character (CONTROL Q) to continue the output, or a BREAK, should then be input.

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 C1000
 ***M JOB X03 XPMGRZS X03FAST P8
 ***M JOB X03 XPMGRZS X03SHORT P8 C10
 ***M JOB X03 XPMGRZS X03LONG C10000 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.

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 21). 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 the specific language chapters in 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 the INDEX.

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)
```

This 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 size ADDR) and PWW0, PWW1, etc. (of size 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 FILE 1 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 it 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 normally result in the current file being used instead. If an input document name is omitted when no current file is defined, or when the current file has already been assigned to another stream, the currently selected input stream is used.

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,I,I) CF

This command copies data from the document specified by P1 to the destination given by P2. P3 gives the type of output document (see Chapter 3), and is interpreted as follows

```
 _______________________________________________________________
|                                                   |  |    |
|___________________________________________________|__|____|
                                                     |   |
      SET TYPE OF OUTPUT DOCUMENT ----               ----    |
                                                         |
            TYPE - 00 CHARACTER         -------------------
                   01 BINARY
                   10 UNIT
                   11 RECORD
```

If no type is provided, the output document will be in the same format as the input document. By specifying a type, this procedure will convert a document between the different types.

P4 and P5 specify starting and finishing positions within the input document. They are in the form as returned by I.POS (see Chapter 4).

Where the output document is defined to be of type unit or record, P6 gives the unit size or maximum record size.

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 9 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, 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 (see below), and may normally be omitted.

P3 has the bit significant encoding.

```
                  |______________|__|__|__|__|__|____| 8 TL MODE BITS|
                                 |  |  |  |  |     |
INHIBIT LSPEC GENERATION -----------  |  |  |  |     |
RUN TIME M/C REAL SIZE                |  |  |  |     |
IS 64 BITS ---------------------------   |  |  |     |
RUN TIME M/C REAL SIZE                   |  |  |     |
IS 32 BITS ------------------------------   |  |     |
SUPPRESS TL CALL ---------------------------   |     |
SUPPRESS TL.END CALL ---------------------------     |
RUN TIME MACHINE SIZE --------------------------------
   00 AS COMPILE TIME MACHINE
   01 16 BIT ADDR 16 BIT INTEGER
   10 32 BIT ADDR 32 BIT INTEGER
   11 32 BIT ADDR 16 BIT INTEGER
```

Normally a compiler will enclose the compiled code in the MUTL statements

```
TL
TLMODULE
.
TLENDMODULE
TLEND
```

It is only in situations where multi module programs/libraries are to be compiled that the TL and TLEND statements need to be suppressed. The TL mode bits are described in Chapter 23 but the main ones are

```
         |_________________|_|_|_|
                            |   |
COMPILE A LIBRARY ----------    |
SUPPRESS RUNTIME DIAGNOSTICS ---
```

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.

Note that if a library is to be used from the command level it may only contain procedures whose parameters are of the types described in 2.3.3. If it is to be used by programs, its procedure parameters may be any basic type or pointer but not a user defined type.

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 precompiled library P1 and links it in to the library directory structure so that its procedures become available as are the system library procedures. P1 is the file name of a compiled library. P2 is the mode in which the library is to be opened. When a library procedure name is given, 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 Chapter 21.

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 into virtual memory on each call, P2 = 1 indicates that the library is to be permanently mapped.

4) RELEASE.LIB ([C]) RL

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

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 IR
insert '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 20, 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                  16:52:49 08/04/81
**NEW
/       INTEGER SIEVE(1000),PRIMES(200),LIM,P,J
/       WRITE(*,200)
/   200 FORMAT(1X,'TYPE PRIMES LIMIT,I3 FORMAT PLEASE')
/       READ(*,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(*,201)LIM
/   201 FORMAT(1X,'PRIMES UP TO ',I3)
/       WRITE(*,202) (PRIMES(I), I = 1,P)
/   202 FORMAT(/(1X,8(I3,4X)))
/       STOP
/       END
/       *END
//
**FORTRAN

MU FORTRAN 3.81  16:59:58  08:04:81

     1.16             WIRTE(*,201)LIM
                          ↑
 **ERROR** INVALID SYNTAX
NO OF ERRORS        1
**EDIT
->C16W
     1.16   )↑          WIRTE(8,201)LIM
->D/IR/I/RI/
->W
     1.16   )        WRI↑TE(8,201)LIM
->E
<CFILE>  AT 17:00:05 ON 08/04/81 OK
**FORTRAN

MU FORTRAN 3.81  16:00:12  08:04:81
**RUN
TYPE PRIMES LIMIT,I3 FORMAT PLEASE
->020

PRIMES UP TO 20

   1     1     1     1     1     1     1     1
```

```
STOPPED:LINE      1. 20

**EDIT
->C/PRIMES(P)/W
     1.12   )¯↑¯         PRIMES(P)  =  1
->D/1/I/I/W
     1.12   )          PRIMES(P)  =  I¯↑¯
->E
<CFILE>  AT 17:00:38 ON 08/04/81 OK
**FORTRAN

MU FORTRAN 3.81  17:00:50  08:04:81
**RUN
TYPE PRIMES LIMIT,I3 FORMAT PLEASE
->020
PRIMES UP TO 20

  2     3     5     7     11     13     17     19



STOPPED:LINE      1.20
**SAVE DEMO
**DELETE
**LISTFILE DEMO
     1.1              INTEGER SIEVE(1000),PRIMES(200),LIM,P,J
     1.2              WRITE(*,200)
     1.3          200 FORMAT(1X,'TYPE PRIMES LIMIT,I3 FORMAT PLEASE')
     1.4              READ(*,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(*,201)LIM
     1.17         201 FORMAT(1X,'PRIMES UP TO ',I3)
     1.18             WRITE(*,202) (PRIMES(I), I = 1,P)
     1.19         202 FORMAT(/(1X,8(I3,4X)))
     1.20             STOP
     1.21             END
     1.22             *END
**STOP
17:01:44 08/04/81 STOP REASON 0
```

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 documents, 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 each 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 OUTCH
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 single pointer 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 implementor 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 STR0* 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 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 initialise 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 * %80	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 %80	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 %180	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) FORTRAN 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.

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.
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 haited. 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 <u>within</u> 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 returns 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 <u>after</u> 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 <u>units</u>. 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 <u>only</u> 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	OUTPUT
SET.I.BPOS (if random)	SET.O.BPOS (if random)
IN.UNIT	sequential character/binary output
sequentialcharacter/binary input	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 accessed 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:

INPUT	OUTPUT
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.

8) SKIP.CHARS(II,[C],I)I

This procedure skips over characters in the currently selected input stream. P1 specifies the end condition as follows:

-1 -	skip to the end of the string specified in P2 or to the terminator character (P3) or to the end of the stream whichever happens first.
0 -	skip to the end of the stream.
>0 -	skip to the beginning of the line specified by P1 (see I.POS for encoding) or end of the stream whichever happens first.

The result applies to skip to string and skip to line cases, where it will be non–zero if the required position is not found.

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, preceded 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,I)

This procedure prints the parameter P1 (which is a LOGICAL32) as a P2–digit hexadecimal number on the currently selected output stream. P2 may be in the range 1 to 8.

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 suppressed completely.

16) COPY.CHARS.OUT(II,[C],I)I

This procedure copies characters from the currently selected input stream to the currently selected output stream. P1 specifies the end condition as follows:

-1 -	COPY to the end of the string specified in P2 or to the terminator character (P3) or to the end of the stream whichever happens first.
0 -	COPY to the end of the stream.
>0 -	COPY to the beginning of the line specified by P1 (see I.POS for encoding) or end of the stream whichever happens first.

The result applies to COPY to string and COPY to line cases, where it will be non–zero if the required position is not found.

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 local 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 PERMIT 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. With no parameters specified, the initial directory is re–selected. The file directory of the subordinate may be selected without giving a 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. The access permission is formed from the five segment access bits (see Chapter 13) combined with the next most significant bit, which if set indicates that exclusive access is required. 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 PW3 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 amount 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
```

PWW1 holds the size of the file in bytes, and PWW2 and PWW3 return the update time and the create time (in seconds) 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.

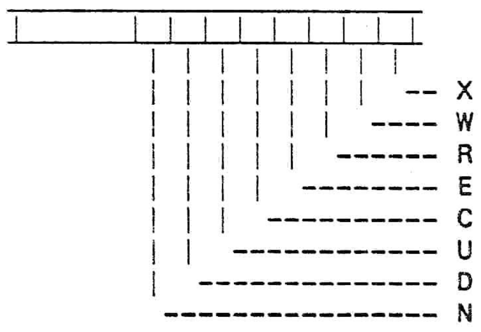
To remove access permission rights, the PERMIT command should be called with P3 zero.

11) CATALOGUE.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 PW2. 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



The number of entries in the segment is returned in PW1.

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

This command provides a mechanism for accessing files from auxiliary discs or 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 required directory. P2 and P3 give the user name and password of the remote directory. If P4 is non-zero the directory will be from the auxiliary disc P4. If P4 is zero the directory will be from the system disc of the selected machine. P5 gives the machine name. If this is zero, the current machine will be assumed.

5.4. FILE EDITING

Two editors are normally available in MUSS, a line editor with limited screen editing capability and a full screen editor. On some small machines the latter may not be available.

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([C],[C],I)   ED

The first 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 third parameter is only relevant if the screen editing command VIEW is to be used. It specifies the type of terminal. A value of zero always indicates a terminal which responds to the ANSI standard sequences for cursor movement and line clearing. The significance of other values is installation dependent.

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

1. <u>Copying</u>	information unchanged from the input file output file.
2. <u>Skipping</u>	over information in the input file, without copying to the output file.
3. <u>Inserting</u>	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.

5.4.1.1. 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.

5.4.1.2. 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.

5.4.1.3. 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 between a pair of string delimiters. Any single non–alphanumeric character 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. !!0C!TITLE! represents a newpage character followed by TITLE.

3. //0A//0A// represents two consecutive newline symbols.

Note 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//

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).

5.4.1.4. 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.

5.4.1.5. 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 page number and its line number. 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'. (Effectively, 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 proceeding, 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 between 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
      /
```

5.4.1.6. 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.

5.4.1.7. 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 VIEW command (V)

The VIEW command causes a specified number of lines from the current position onwards in the input stream to be made available for screen editing. Up to 23 of these will be displayed initially with a cursor positioned at the top lefthand corner. In the case where the View command specifies more than 23 lines, the remainder will roll on to the screen if an attempt is made to move the cursor beyond the end of the screen. The V command has the same form as the Window command.

i.e. V integer e.g. V30

In certain circumstances the amount of text made available for editing will be less than requested and a warning message will be given indicating where it has been curtailed. The reasons for this are:

1) The occurrence of a line in the input that is too wide to display across the screen. In this case the text for editing will stop at the start of the long line and the input pointer will point to the first character of the long line.

2) The occurrence of a newpage character. The text will be allocated up to the newpage character. It will not be sent with the text but on exit from view it will be transmitted to the output stream.

3) The requested text being larger than the buffer space available to the editor. The text allocated will stop at the end of the last complete line prior to the buffer becoming full.

After the text has been displayed editing is achieved by the set of commands described under VEDIT in section 5.5.

The normal exit from View is achieved by the Control 'E' command which arranges that the edited output is sent to the output file with the input pointer at the end of the string of input just viewed.

An alternative form of exit allows the output to be discarded and the input pointer to be restored to its value prior to the View. This is achieved by Control 'W'.

Two other forms of exit are provided, mainly for clerical use. One of these Control 'Z', is to allow a file to be searched a section at a time until a specific feature is seen. It allows the output to be discarded but the input to be stepped on past the section requested for viewing. The final form of exit is Control 'A' which allows a template document to be successively output with different alterations each time. This is achieved by sending the edited output to the output stream while leaving the input pointer where it was prior to viewing. This allows the same section of input to be viewed and edited any number of times.

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.

5.4.1.8. 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 5 though this should rarely be necessary.

Example

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

5.5. A BASIC SCREEN EDITOR

VEDIT([C],I,I,I)I VED

This is the limitud screen editor used by the View command above and some other system utilities. The first parameter describes the vector containing the text for screen editing. The second parameter gives the size of the text for editing which is planted at the start of the vector leaving a space at the end. This is required for insertions of new data. The third parameter describes the device interface type (0–ANSI, 1–Installation dependent). The fourth parameter is a stream number. This is the overflow output stream, which is used when too many characters have been inserted and the text buffer is full. When this occurs the buffer is flushed sending all characters prior to the current cursor position to the output stream. This action is not evident to the user as the screen remains unaltered but from then on the cursor cannot be moved prior to this point. VEDIT returns the effective size of the vector after editing when all the edited text appears from the start of the specified vector. This is followed immdiately by the exit command type, which provides the opportunity for the calling process to take special action on the edited text.

5.5.1. Commands

Users of ANSI standard terminals may use the special keys associated with the following actions or the control alternatives. Other users must use the control alternatives.

	Control Code
Cursor Up	Control U
Cursor Down	Control Y
Cursor Right	Control P
Cursor Left	Control O
Delete Character	Control D
Delete Remainder of Line	Control X
Command Exit	Control E,A,W,Z

Insert Space	Control R

Any other characters are assumed to be replacements. They will not only obliterate those already there on the screen but will also delete them from the text. Space for an insertion is made available by use of the Control R command.

5.6. THE SCREEN EDITOR

SEDIT([C],[C],I) SED

This command invokes a general purpose screen editor with a large set of screen editing facilities.

The first two parameters are filenames. P1 is the input file and P2 the output file. If P1 is zero, the current file is used as input. However, if current file is not available the editor assumes that the user has asked for a new file to be created and consequently allows the user to insert data (and edit it).

P2 is the destination file which can be any filename or the current file (0) and will contain the edited (or newly input and edited) data. P3 is the terminal type.

The display editor divides the screen into two areas, the biggest area (at the top) is the one in which the text appears and editing is done. The bottom few lines are reserved for displaying commands or any other informative messages.

All the editing operations are performed on programs and English texts which are displayed on the screen. The editor has short and simple commands which show their effects in the display as soon as they are typed in. Text (printing and formatting characters) is inserted just by typing it; there is no "insert" command.

Generally, each screen line contains a separate line of data. However, if a dataline is too long it will be displayed on several screen lines. Most edit functions (e.g. delete line) are restricted to within the screen line and not the actual line.

The editor provides seven different types of operations: cursor control, insertion, deletion, screen control, mark and region, kill/unkill and miscellaneous. The form of these commands and their characteristics are described in the following sections.

5.6.1. Form of Commands

Most of the frequently used commands are entered using a single key This is either in the form of a control character or in the case of an ANSI standard terminal the corresponding specially provided key. The main commands for which special keys are normally provided are given below with the control character equivalents. Some additional commands only have control character representations and these are introduced in the sections where they are described.

OPERATION	ANSI	ALTERNATIVE CONTROLS	
cursor up	ESC[A	CTRL-U	%15
cursor down	ESC[B	CTRL-Y	%19
cursor forward	ESC[C	CTRL-P	%10
cursor backward	ESC[D	CTRL-O	%F
delete line	ESC[M	CTRL-X	%18
delete next character	ESC[P	CTRL-D	%4
scroll up	ESC[S	CTRL-A	%1
scroll down	ESC[T	CTRL-Z	%1A
previous page	ESC[V	CTRL-B	%2
next page	ESC[U	CTRL-V	%6

There are also some commands such as mark and region operations which are double characters. That is, two characters must be typed consecutively with the CTRL held down. After entering the first one users can still change their minds and start insertion or type any non–existing command to cancel the effect of the double command.

5.6.1.1. Summary of Commands

There are seven categories of operations each with a set of commands which are outlined here and will be described individually in greater detail later.

For the rest of this section, commands are only referred to by their control character identifier. Users may use ANSI commands whenever provided instead of these controls (see 5.6).

(C – means, CTRL key held down).

A. Cursor Control Commands

1. C–W	Move cursor to the beginning of a line.
2. C–E	Move cursor to the end of a line.
3. C–P	Move cursor forward by one or more character positions.
4. C–O	Move cursor backward by one or more character positions.
5. C–Y	Move cursor down by one or more lines.
6. C–U	Move cursor up by one or more lines.

B. Insertion

No special command, just type in. Receipt of commands will cause the line/screen to be refreshed.

C. Deletion Commands

1. C–D	Delete one or more following characters.
2. C–T	Delete one or more preceding characters.
3. C–X	Kill all or part of the current line.

D. Screen Control Commands

1. C–Z	Scroll down by two or more lines.
2. C–A	Scroll up by two or more lines.
3. C–V	Scroll up one or more pages.
4. C–U	Scroll down one or more pages.

E. Mark and Region Commands

1. C–G C–B	Set mark.
2. C–G C–V	Interchange mark and cursor.

F. Kill/Unkill Commands

1. C–G C–Z	Save region.
2. C–G C–X	Kill region.
3. C–G C–A	Get back (unkill).

G. Other Commands

1. C–R	The following integer is the repeat counter.
2. C–G C–R	Display repeat counter.
3. C–G C–W	Quit (abort) edit.
4. C–G C–E	Exit (update and write).

5.6.1.2. Numeric Arguments

Many screen editing commands can be given a numeric argument. Some commands interpret the argument as a repetition count. In all cases, no argument is equivalent to an argument of one.

An argument can be entered by typing C–R followed by an unsigned integer. The next non–digit is assumed to be the command for which the argument was intended. However, if the following non–digit is not a command, the argument will be reset.

To cancel or change an argument another C–R <digits> may be typed.

The effect of a numeric argument (n) on each command is described in the following sections.

5.6.1.3. Cursor Control Commands

The real position of the cursor on the screen is between the position at which it resides and the one before it and hence 'next' and 'previous' positions are identified.

The line within which the cursor resides at any time is called the 'current line'.

Cursor control commands only operate within the text area and will not move to positions outside the displayed text.

1) Beginning of the Line (C–W).

This command positions the cursor at the first character position of the line.

2) End of the Line (C–E).

This command moves the cursor to the end of the current line and positions it over (i.e., before) the last character on a screen line.

3) Cursor Forward (C–P).

This command moves the cursor forward by (n) character positions. If the cursor is at the end of a line, it will be moved to the beginning of the next line. scrolled up first so that the required line is within the screen.

4) Cursor Backward (C-O).

This command moves the cursor backward by (n) character positions. If the cursor is at the beginning of a line, it will be moved to the end of the previous line.

5) Cursor Down (C-Y).

This command moves the cursor to a similar position within the n-th following line from current line.

Operation stops as soon as the required line is reached or end of the file is encountered in which case the cursor will be positioned within the last line.

If the new position is beyond the end of line (EOL) marker, the cursor will be repositioned at the EOL marker.

6) Cursor Up (C-U).

This command moves the cursor to a similar position within the n-th preceding line from current line.

If the new position is beyond the EOL marker, the cursor will be repositioned at the EOL marker.

5.6.1.4. Insertion

To insert printing characters into the text, just type them in at an appropriate position. This causes any non-control character to be echoed on the screen while all characters are buffered for transfer to the editor. Upon receipt of any non-insertable character (e.g. editing commands), all or part of the screen will be refreshed by the editor. This will cause newly inserted control characters (e.g. VT) to be displayed as well. As a result users may type in as much data as they wish (even past the text area of the screen). However, receipt of any command or non-insertable character will initiate the refreshing of the screen before the command itself is serviced.

Sometimes it may be necessary to clear the screen and refresh it, this can be done by issuing a screen control command (5.6.1.6). Format effectors such as linefeed, formfeed, carriage-return, vertical tab, horizontal tab and backspace can be inserted using their corresponding key (or CTRL + key). When displayed, they will be shown as c.

If too many characters are added to one line without breaking it with a carriage return (CTRL-M), linefeed (CTRL-J) or formfeed (CTRL-L), the line will grow to occupy two (or more) lines on the screen.

If the new position of the cursor after insertion falls outside the text area scope, the whole of the text will be scrolled up in order to show the line which contains the cursor at the centre.

5.6.1.5. Deletion Commands

All deletion operations are strictly limited to the data being displayed at any time. Only kill (delete) line saves the data being removed, which may be restored using GET BACK command (5.6.1.8).

7) Delete Following Character(s) (C-D).

This command deletes the next character and all remaining characters are shifted to the left by one character position. Position of the cursor is not affected by this command.

8) Delete Previous Character(s) (C–T).

This command deletes the preceding character and moves the cursor and the following characters backward by one character position.

9) Kill/Delete Line (C–X).

Operation of this command depends on the value of the numeric argument.

If n is zero, all characters of the current line before the cursor are deleted and the cursor and rest of the line are shifted to the left.

If n is non–zero, only the current line is affected. In this case all characters following the cursor up to but excluding the EOL marker (if any are removed). If there is no EOL, only contents of the current line on the screen are removed. With the cursor positioned at the EOL, the EOL itself is removed. Consequently, in order to delete one line the C–X has to be used twice.

5.6.1.5.1. Deleting Long Lines

Long lines may be deleted by positioning the cursor at the beginning of each portion of the line on the screen and using C–X.

If all but the first screen line of a long line are deleted the line still continues to be a long line until a separator (eg.,LF) is inserted at the end of the line.

5.6.1.6. Screen Control Commands

The text or edit area of the screen always shows a limited amount of text and data, starting from a certain point within the editor's buffer. This area, therefore acts like a 'window' to the buffer. The window may be moved up and down in order to change the active area (on which editing is done). Notice that because of the automatic wrapping which is performed on long lines, there is no need to move the window sideways.

10) Scroll Down (C–Z).

This command moves start of the window down by n(2 or more) lines, unless end of the text is reached before that.

Position of the cursor at the end of the operation depends on the n. Generally the cursor moves up with the text until it reaches the top of the screen. At this point any further scroll causes it to be positioned at the top left hand corner of the screen.

To read a long file sequentially this command can be used without any parameter. It takes the last two lines at the bottom of the text area and puts them at the top, followed by many more lines not visible before.

If n = 0, it clears the screen before refreshing again.

11) Scroll Up (C–A).

This command moves start of the window up by n lines, unless beginning of the file is reached before that.

Position of the cursor at the end of the operation depends on n. Generally the cursor moves down with the text until it reaches the bottom of the text area. At this point an further scroll causes it to be positioned at the beginning of the last line.

This command with no parameter can be used for going back through the file sequentially. It takes the top two lines and puts them at the bottom of the text area. The rest is filled with the preceding lines in the file.

If n = 0, it clears the screen before refreshing again.

12) Page Up (C–B)

This command repositions start of the window at the beginning of the n preceding pages and displays.

If n = 0, it moves to the beginning of the file.

13) Page Down (C–V)

This command positions start of the window at the beginning of the n following pages and displays.

If n = 0, it moves to the end of the file.

5.6.1.7. Mark and Region Commands

In general, a command which processes part of the input text must be able to be told where to start and where to stop. Here, such commands (see 5.6.1.8) start at the cursor position and end at a place called the "mark". This range of text is called the "region".

For example, to remove part of a text, user could first set go to the beginning of the text to be removed, put the mark there, move to the end and then issue the KILL command. Or set the mark at the end of the text, move to the beginning, and then issue the command.

The maximum gap size between mark and cursor is machine–dependent. On most machines this will be about the same size as the segment size. If the mark becomes undefined, any operations related to it will cause an appropriate message to be displayed.

17) Set Mark (C–G C–B).

This command places a mark where the cursor is currently positioned. In fact the mark is invisible and only its position within the file is remembered.

18) Interchange Mark and Cursor (C–G C–V).

This command puts the mark where the cursor was and cursor where mark was. Thus, the previous location of the mark is shown, but the region is not changed.

Users in order to reassure themselves that the mark is where they think it is, can use this command.

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. There are also some output procedures for printing a dictionary or index. 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 daisy wheel printers and ordinary printers or terminals. All printers are assumed to have a full visual character set but the daisy wheel printers are also assumed to have

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

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 and serves as an example of the facility. The formal specification of TEXT is

```
          TEXT ( [C],[C],II,[C],[C] )
    where P1 specifies the input file
          P2 specifies the output file
          P3 = 0 ordinary printer (LPT)
             = DBL diablo printer
             = LPT line printer
          P4 = specifies a contents file
          P5 = specifies an index file
```

If P4 and P5 are omitted the contents and index information are suppressed. The contents file when produced will contain entries for all Chapter and Section headings (i.e. @R, @S1, @S2, @S3 described below) in TEXT format with page numbers added.

The index file will be an unsorted list of words and page numbers. It is normally expected that a document will be produced a Chapter at a time, and that the index files will be edited together and listed using the facilities of 6.1.7.

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 R indicates the start of a new chapter. It should be followed by an integer giving the chapter number and a string giving its name. The style of output will depend upon the facilities of the printer. An entry will also be generated in the contents file.

@ followed by S1, S2 or S3. This indicates the start of a new chapter or 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 heading. There is no distinction between S1, S2, S3 from the point of view of the formatted text. However, they cause different degrees of indentation and separation in the contents file. S1 should be used for chapter headings (as an alternative to @R), S2 for sections and S3 for subsections.

@ 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 O. This is again concerned with underlining. It 'toggles' an 'underline switch'. The first use of it will 'turn on' the underlining of all following characters until a second use of it turns the underlining off. It should not be used in conjunction with @C, W, L.

@ followed by K. This is used for highlighting changes by means of margin bars. The first occurrence of @K causes a switch to be set. Whilst this switch is set all lines on that will be preceded by '|' in the margin. A second occurrence of @K resets the switch and the lines which follow will not be highlighted, unless a further @K occurs to set the switch again.

@ followed by X. In addition to '@' three other symbols, namely 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. For parameter values up to ten, 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. Parameters greater than 10 relate to strings, hence any integer pair in the above sequence may be replaced by an integer greater than 10 and a string terminated by a space or newline. 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
8 chapter number
9 page number
11 page heading

Four 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 for parameter 8 the chapter component of a page identification will be omitted. If a negative number is given as the page number, pages will not be numbered at all.

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'.

Any string specified as a page heading will appear at the start of each subsequent page, for example the "MUSS USER MANUAL" above.

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 which are to be 'glued' into place,' the 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 inlines. If there is not enough room on the current page a new page will be started and the requested space will be left blank at the head of it. A further alternative which leaves the requested amount of space on the current page if possible otherwise at the top of the next page, whilst still utilising the current page for subsequent text, 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. The simplest 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. A variant of this is @U which in addition to operating like @Q suppresses all layout changes for the specified number of lines. 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 be output immediately if there is room, otherwise if it 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 @H which causes the line of text following it to be output in large letters. The style is device dependent and simple character devices will give

HEADING

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
@H	heading
@I	indentation

@J	add to index
@K	highlight by margin bars
@L	underline one line
@M	line centreing
@N	indentation following line centreing
@O	toggle the underline switch
@P	new page
@Q	force new page for large tables
@R	chapter heading
@S	section start
@T	tabulation stops
@U	user formatted diagram
@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. A suitable print procedure is given in 6.1.7.

SPELL (P1, P2)
P1 is a dictionary file name
P2 is a text file name.

6.1.7. Listing Facilities

Three special listing procedure are provided. Two of these allow dictionaries and document index to be sorted and listed, the third allows a complete module to be 'TEXT'ed and drawn in both English and MUSL on the lineprinter.

1) LIST.DICT([C],[C])

This procedure sorts the dictionary contained in the file specified by P1 and outputs the result to P2. The output has four words per line sorted into alphabetical order with a blank line separating words with a different initial letter.

2) LIST.INDEX([C],[C])

This procedure lists the index contained in P1 in stream P2. An example of the format produced is the index for this Manual.

3) LIST.MOD([C])

This procedure firstly uses TEXT to output to the lineprinter the descriptive text at the start of the file P1 and follows this by the flowcharts drawn at both levels 0 and 1.

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 diablo printer, for example). The chart descriptions may of course be edited, and the charts re–drawn as necessary.

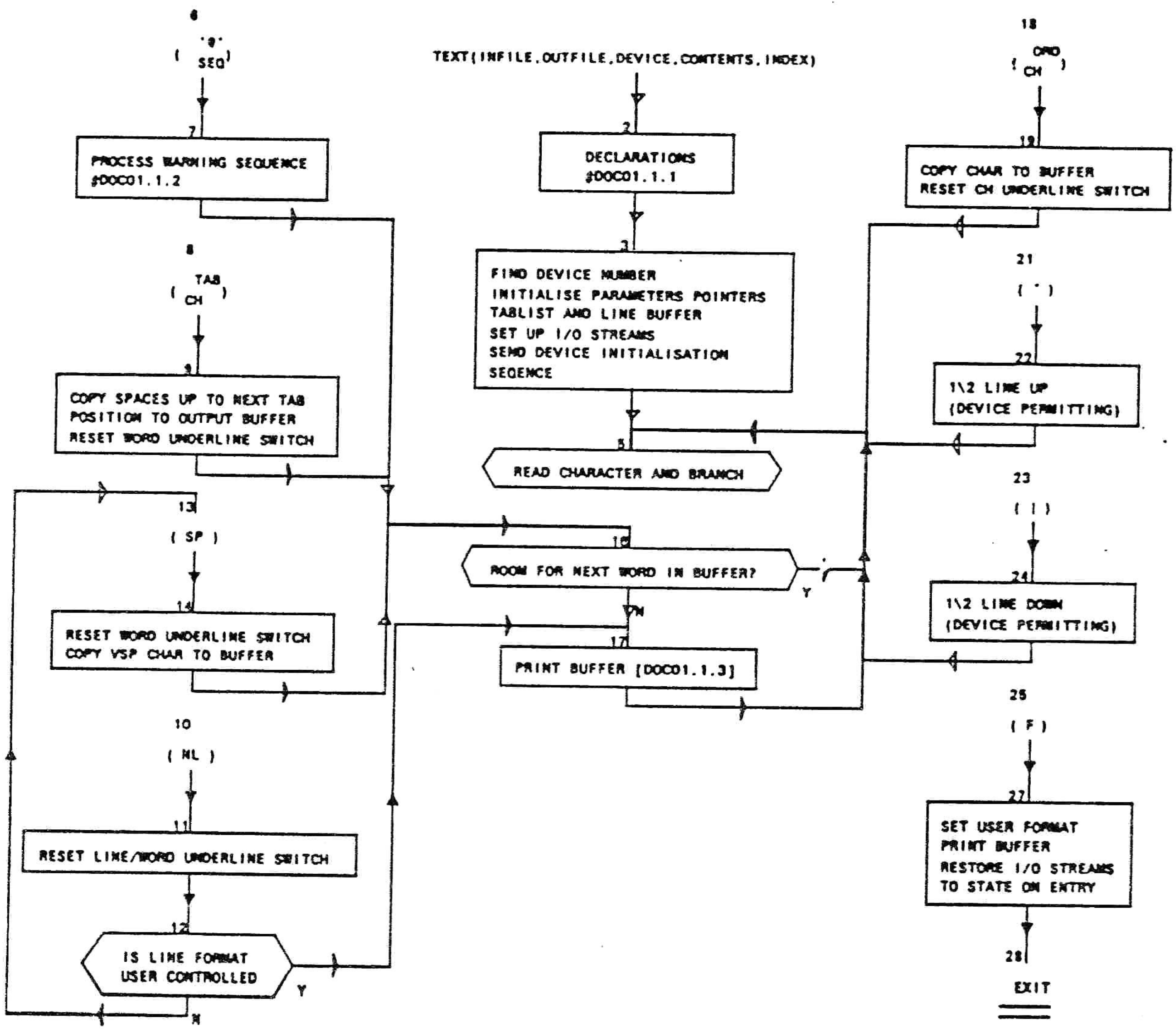
To enable Flocoder to generate or display the required program, the user provides a 'translation' for each box. If the action required in a box is simple, it will translate into a sequence of statements in a programming language; if it is complex, the translation may reference other flowcharts. In this 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

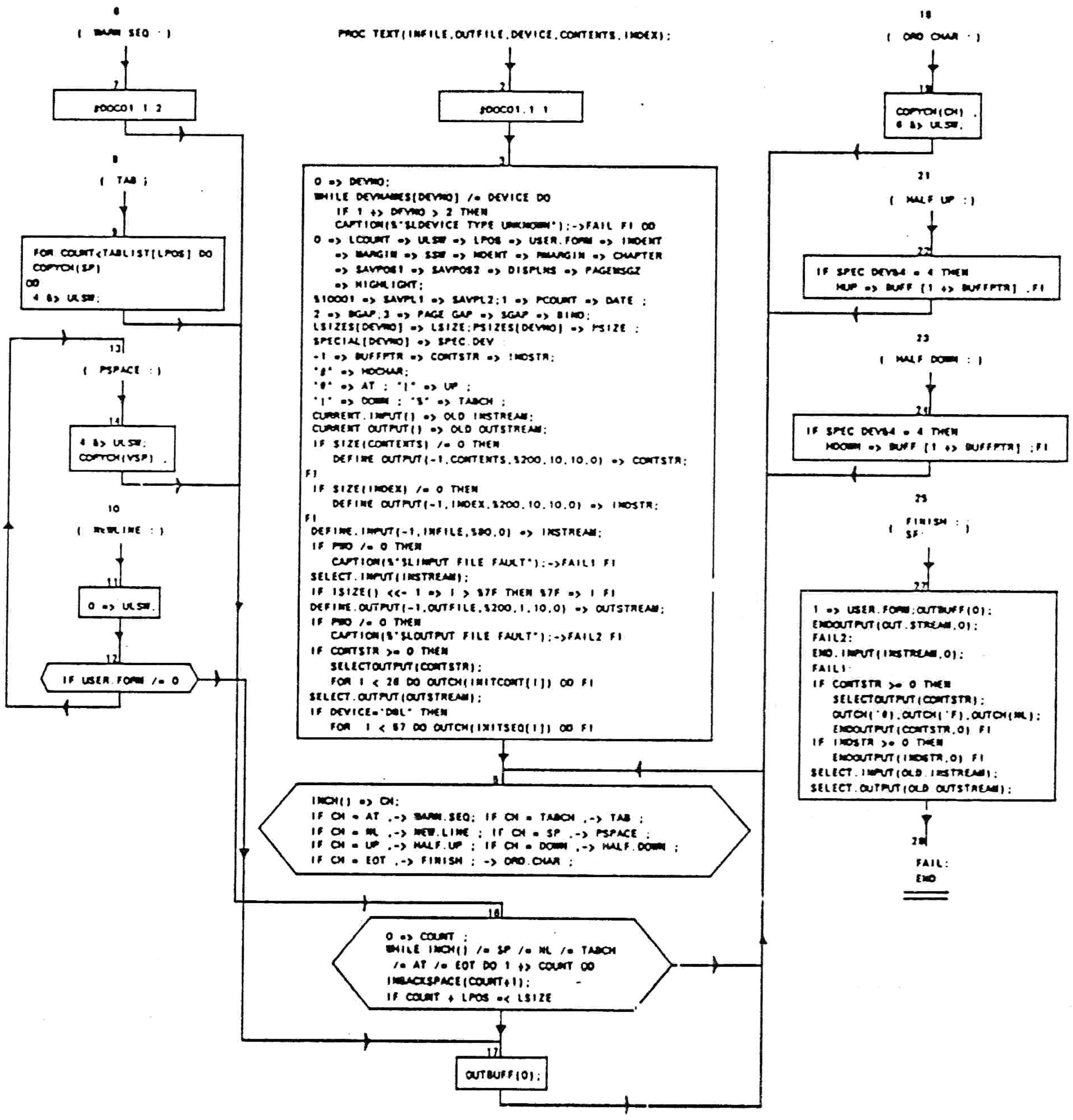
A TITLE statement
One or more COLUMN statements
Zero or more ROW statements
Zero or more FLOW statements
One or more BOX statements
An END statement.

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



6
('@' SEQ)
7
PROCESS WARNING SEQUENCE #DOCO1.1.2
TEXT(INFILE,OUTFILE,DEVICE,CONTENTS,INDEX)
2
DECLARATIONS #DOCO1.1.1
3
FIND DEVICE NUMBER INITIALISE PARAMETERS POINTERS TABLIST AND LINE BUFFER SET UP I/O STREAMS SEND DEVICE INITIALISATION SEQENCE
5
READ CHARACTER AND BRANCH
18
(ORD CH)
19
COPY CHAR TO BUFFER RESET CH UNDERLINE SWITCH
8
(TAB CH)
9
COPY SPACES UP TO NEXT TAB POSITION TO OUTPUT BUFFER RESET WORD UNDERLINE SWITCH
21
(^)
22
1\2 LINE UP (DEVICE PERMITTING)
13
(SP)
14
RESET WORD UNDERLINE SWITCH COPY VSP CHAR TO BUFFER
16
ROOM FOR NEXT WORD IN BUFFER?
Y
N
17
PRINT BUFFER [DOCO1.1.3]
23
(|)
24
1\2 LINE DOWN (DEVICE PERMITTING)
25
(F)
27
SET USER FORMAT PRINT BUFFER RESTORE I/O STREAMS TO STATE ON ENTRY
28
EXIT
10
(NL)
11
RESET LINE/WORD UNDERLINE SWITCH
12
IS LINE FORMAT USER CONTROLLED
Y
N

Figure 6-1 DOCO1.1 Level 0



```
PROC TEXT(INFILE,OUTFILE,DEVICE,CONTENTS,INDEX);

2
#DOCO1.1 1

3
0 => DEVNO;
WHILE DEVNAMES[DEVNO] /= DEVICE DO
   IF 1 +> DEVNO > 2 THEN
   CAPTION(%"$LDEVICE TYPE UNKNOWN");->FAIL FI OD
0 => LCOUNT => ULSW => LPOS => USER.FORM => INDENT
   => MARGIN => SSW => NDENT => RMARGIN => CHAPTER
   => SAVPOS1 => SAVPOS2 => DISPLNS => PAGEMSGZ
   => HIGHLIGHT;
%10001 => SAVPL1 => SAVPL2;1 => PCOUNT => DATE ;
2 => BGAP;3 => PAGE GAP => SGAP => BIND;
LSIZES[DEVNO] => LSIZE;PSIZES[DEVNO] => PSIZE ;
SPECIAL[DEVNO] => SPEC.DEV
-1 => BUFFPTR => CONTSTR => INDSTR;
"£" => NOCHAR;
"@" => AT ; "|" => UP ;
"|" => DOWN : "$" => TABCH ;
CURRENT.INPUT() => OLD INSTREAM;
CURRENT OUTPUT() => OLD OUTSTREAM;
IF SIZE(CONTENTS) /= 0 THEN
   DEFINE OUTPUT(-1,CONTENTS,%200,10,10,0) => CONTSTR;
FI
IF SIZE(INDEX) /= 0 THEN
   DEFINE OUTPUT(-1,INDEX,%200,10,10,0) => INDSTR;
FI
DEFINE.INPUT(-1,INFILE,%80,0) => INSTREAM;
IF PWO /= 0 THEN
   CAPTION(%"$LINPUT FILE FAULT");->FAIL1 FI
SELECT.INPUT(INSTREAM);
IF ISIZE() <<- 1 => I > %7F THEN %7F => I FI
DEFINE.OUTPUT(-1,OUTFILE,%200,1,10,0) => OUTSTREAM;
IF PWO /= 0 THEN
   CAPTION(%"$LOUTPUT FILE FAULT");->FAIL2 FI
IF CONTSTR >= 0 THEN
   SELECTOUTPUT(CONTSTR);
   FOR I < 28 DO OUTCH(INITCONT[I]) OD FI
SELECT.OUTPUT(OUTSTREAM);
IF DEVICE="DBL" THEN
   FOR  I < 87 DO OUTCH(INITSEQ[I]) OD FI

5
INCH() => CH;
IF CH = AT ,-> WARN.SEQ; IF CH = TABCH ,-> TAB ;
IF CH = NL ,-> NEW.LINE ; IF CH = SP ,-> PSPACE ;
IF CH = UP ,-> HALF.UP ; IF CH = DOWN ,-> HALF.DOWN ;
IF CH = EOT ,-> FINISH ; -> ORD.CHAR ;

6
( WARN SEQ : )
7
#DOCO1 1 2

8
( TAB ;
9
FOR COUNT<TABLIST[LPOS] DO
COPYCH(SP)
OD
4 &> ULSW;

13
( PSPACE : )
14
4 &> ULSW;
COPYCH(VSP) .

10
( NEWLINE : )
11
0 => ULSW.
12
IF USER.FORM /= 0

16
0 => COUNT ;
WHILE INCH() /= SP /= NL /= TABCH
 /= AT /= EOT DO 1 +> COUNT OD
INBACKSPACE(COUNT+1);
IF COUNT + LPOS =< LSIZE

17
OUTBUFF(0);

18
( ORD CHAR : )
19
COPYCH(CH) .
8 &> ULSW.

21
( HALF UP : )
22
IF SPEC DEV&4 = 4 THEN
   HUP => BUFF [1 +> BUFFPTR] .FI

23
( HALF DOWN : )
24
IF SPEC DEV&4 = 4 THEN
   HDOWN => BUFF [1 +> BUFFPTR] ;FI

25
( FINISH :
  SF         )
27
1 => USER.FORM;OUTBUFF(0);
ENDOUTPUT(OUT.STREAM,0);
FAIL2:
END.INPUT(INSTREAM,0);
FAIL1:
IF CONTSTR >= 0 THEN
   SELECTOUTPUT(CONTSTR);
   OUTCH('@).OUTCH('F).OUTCH(NL);
   ENDOUTPUT(CONTSTR,0) FI
IF INDSTR >= 0 THEN
   ENDOUTPUT(INDSTR,0) FI
SELECT.INPUT(OLD.INSTREAM);
SELECT.OUTPUT(OLD OUTSTREAM);

28
FAIL:
END
```

Figure 6-2 DOCO1.1 Level 1

6.2.1. The TITLE Statement

Example:

```
@TITLE DOC01.1.1(1,7)
```

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 and may optionally include in round brackets version and generation numbers. By convention, the characters 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-5T-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 (double underlined)
N	Null box (a point)
R	Rectangle box
S	Start box (no outline)
T	Test box

6.2.3. The ROW Statement

Example:

```
@ROW 6-1-18
```

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–13–14–16N–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 17.0
PRINT BUFFER[DOC01.1.3]
@BOX 17.1
OUTBUFF(0);
```

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' may be 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 language (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.

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 this subchart will be inserted in DOC01.1.1, at the specified place, whenever code is generated for it. Note that the bracketed version and generation numbers are not used in cross–referencing which is implemented by using a hash of the title characters up to either left round bracket or newline. For further details read the implementation description.

In some languages (e.g. Pascal) it is necessary to declare the labels which are used. Therefore a variant of the cross reference notation is provided for this purpose if #:chart name is used it will be replaced by the list of label names used in the given chart separated by commas.

6.2.6. The END Statement

Example:

```
@END
```

This statement terminates the description of a flowchart.

6.2.7. Flocoder Procedure Specifications

There are five 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. 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 which is generated becomes 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.

P3 – This is the Print switch which controls monitoring on the currently selected output stream. If this is = 0 minimal monitor printing occurs if it is /= 0 the code for each box at the specified level is also listed.

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 which are suitable for MUSL programs are assumed:

```
P4 - L: giving labels of the form LXXXXX:

P5 -  -> L; giving gotos of the form -> LXXXXX;

P6 - ,-> L; giving conditionals of the form, -> LXXXXX;
                   newline.
```

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

This procedure is used when a batch of diagrams are to be drawn on a non-interactive output device. It produces output on the file P2 to draw selected charts from the input file P1 on a graphical output device. A list of titles for the required charts should appear on the currently selected input stream, separated by newlines and terminated by the character '@' 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. 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 – name of the output file. Zero means the current file.

P3 – the required level number, as with FLIP, except –1 may be specified, in which case each diagram will be drawn twice at level 0 and level 1.

P4 – 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.

This determines the format of the output on stream P2.

3) DISPLAY([C],[C],I)

This procedure is used from an interactive terminal when it is required to inspect a single flowchart. P1 gives the file name and P2 gives the flowchart name. P3 indicates the code level which is to be displayed. It may be –1, in which case only the box shapes, containing their numbers instead of code, will be displayed. If the diagram is too big to fit on the terminal screen, as much as possible from the lefthand corner will be displayed. After a particular diagram has been selected by means of this command, it may be operated upon by the next two procedures.

4) SCROLL(I,I)

This procedure allows the display to scroll the diagram previously selected by the DISPLAY command. Parameter P1 is normally a letter

D meaning down
U meaning up
R meaning right
L meaning left

If P1 is zero the display is re–initialised from the top lefthand corner. P2 is an integer which specifies the number of lines or character positions over which the scroll is to take place. If P2 is zero the scroll will take place in the direction specified up to the edge of the diagram.

5) LEVEL(I,I)

This procedure allows the displayed level of box P1 to be changed to that given by P2. If P1 is zero the displayed text level of all the boxes will be changed to the new level. If level –1 is specified only box shapes, containing box numbers, will be displayed.

6) FLED([C],[C],I)

This command is used when it is required to screen edit a flocoder file. The file specified by P1 is edited to the new file specified by P2. P3 specifies the device type (0 – ANSI standard, /= 0 – Installation Dependent). Editing is performed by the following commands.

a) T title. The title command allows a flowchart title to be specified, and it causes the input file to be searched (forwards) until the chart is found. At this point the chart is displayed, at level −1. Whilst the search is proceeding the content of the input file is copied to the output file. The chart thus selected is available for editing by means of the commands described below. If a selected chart is edited, and then a further T command is used, the edited form of the chart will be substituted for the original one in the output file.

b) S letter integer. This command calls the SCROLL procedure and the letter specifies the direction of the scroll (U,D,R,L). The integer specifies the distance in characters or lines.

c) L integer integer. This command calls the LEVEL procedure to change the displayed level of a box. The two integers are then used as the parameters of the procedure call, hence they must obey the conventions of LEVEL.

d) M integer integer. This command selects a box for modification. The parameters specify box number and level as in c). The text for the specified box/level is displayed on the screen and may then be screen edited using the conventions described for the Vector Editor (VEDIT – Section 5.5). Any of the four forms of exit from the editor cause a return to the FLED command sequence but 'Control E' also redisplays the chart with the specified box at the specified level.

e) B first/column spec/last. This BOX statement is the means by which new boxes may be introduced into an existing column of the currently selected flowchart. Only one of 'first' and 'last' are required. They are intended to be the existing boxes after or before which the new boxes are to be inserted. The new boxes are specified inside / / in the same way that a column is specified in a flowchart encoding. For example

```
B/21T–22R/11
```

would insert a test box numbered 21, and a rectangle box numbered 22 into a column containing box 11. The insertion would be made immediately above box 11, whereas the statement

```
B 10/21T–22R/
```

would cause the insertion to be after box 10.
A further variant of the 'B' statement allows boxes to be deleted. It is

B – box list. For example,

```
B – 12–16–21
```

will cause boxes 12, 16 and 21 to be removed from the currently selected diagram. The primary flow pattern will be maintained as if the deleted boxes had become NULL boxes. If any of them are test boxes the secondary flows will be lost.

f) C first/column spec/last. This Column statement allows a new column to be introduced. The first and last parameters should specify existing box numbers in the columns to the left and right of the new column. Again only one is required.

g) F flow spec. This Flow statement allows new flowlines to be introduced. For example in the case of the box insertion given above the following flows may have to be added

```
F 10–21N–22–11
F 21YES–11
```

However, if there was previously a primary flow from 10–11 a warning will be given that this has now been destroyed. The flow 10–11 could be explicitly deleted by the statement

F– 10–11

h) R row spec. New row alignment can be forced by means of this statement. Old ones may be removed by the negative form

R– rowspec

i) E. This command ends the edit. If the currently selected chart has been modified, the modified form will be copied to the output file followed by the rest of the input file. Otherwise the unmodified chart and the rest of the input will be copied to the output file.

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 – System Library errors.

7.3. Program Faults

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

7.3.1. 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.

7.3.2. 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.

7.3.3. 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.

7.3.4. Organisational Command Errors

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

7.3.5. Input/Output setup error

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

7.3.6. 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.

7.3.7. Job control command errors

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

7.3.8. 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.

7.3.9. System Library Errors

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

7.4. 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.

6) RETURN.FROM.TRAP( )

This procedure enables a user trap procedure to reset the process to the state it was in just before the trap occurred. The process then continues from the reset point.

7.5. PROCEDURES FOR FAULT MONITORING.

1) OUT.M(I,I)

This procedure outputs, on 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).

7.6. 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.

7.7. 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:

TL.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

TL.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.

The TL.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.

S Surrounding : move to next procedure/block in the static chain.

C Current : move back to the procedure in which the fault occurred.

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

G {<page>.}<line> Go : restart the program at the specified line. If page is omitted, the current page is assumed. Zero indicates the current line.

I {<page>.}<line> Insert : a breakpoint is inserted at the specified line.

R {<page>.}<line> Remove : the breakpoint, at the beginning of the specified line, is removed.

P Profile : the profiles and stored tracebacks are printed out.

E Enter : to re-enter the program after executing the breakpoint.

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.

The value of a variable (or any of its components) will be printed out when <variablename> is typed in, followed by any of the following, indicating the required component.

```
↑             to dereference a pointer
[ ] or ( )      for all elements of a vector
[n] or (n)    for the nth element
@             for all fields of an aggregate type
<fieldname>   for one field only.
Spaces must be omitted before the character
↑, [, (, ], ), @.
```

To alter the value of a variable (or component), the above sequence is followed by /<value>. In the case of a vector for which only certain elements need altering, * is typed for each element unaltered.

* 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.

8. SEMAPHORES

The primary synchronisation mechanism in MUSS is its message system. This allows collaborating processes to be distributed across a network. However, it is possible for processes in the same machine to share segments, providing the memory management hardware allows multiple virtual addresses to be associated with the same physical address. The common segments are one example of this, also when the underlying hardware architecture is suitable, two or more processes may open the same file and share one physical copy. Thus some special applications may run as multiprocess systems with the processes communicating through shared store. The following set of commands are provided, in order to facilitate the synchronisation of these processes. They will not normally be present in the standard general purpose MUSS system, but can be compiled into any special purpose system which require them.

8.1. COMMANDS

(1) CREATE.SEMAPHORE(I)

This procedure creates a semaphore with the number given by P1 for use in the current process. Normally up to 8 semaphores may be used as P1 is taken modulo 8.

(2) PASS.SEMAPHORE(I,I,I)

This procedure is used to enable a semaphore of the current process to be made available to any process created by the current process. P1 gives the semaphore number in the current process, P2 gives its number in the recipient process and P3 gives the SPN of the recipient process. When a semaphore has been passed in this way the recipient and creating processes may both use the P and V operations described below.

(3) INIT.SEMAPHORE(I,I)

This command may only be issued by the processes which created the semaphore P1. It initialises the semaphore value to P2 and sets its wait queue empty.

(4) P.SEMAPHORE(I)

This command performs the Dijkstra "P" operation on the semaphore P1

```
i.e. value(P1) := value(P1)-1
     WHILE P1 < 0 WAIT
```

Thus the current process may be placed in a waiting state until some other processes cause the semaphore value to become $\geq$ 0. Any number of processes may be waiting on the same semaphore and they are ordered in arrival order.

(5) V.SEMAPHORE(I)

This command performs the Dijkstra "V" operation on the semaphore P1

```
i.e. value(P1) := value(P1)+1
     IF P1 ≤ 0 THEN FREE FIRST WAITING PROCESS.
```

8.2. RELEASING SEMAPHORES

When any process ends which is connected to a semaphore the number of users of the semaphore is decremented by 1. If there are other processes still connected to the semaphore it will remain in operation. If the terminating process was in the P state a V operation will be forced. When the terminating process is the only process connected to a semaphore it will cease to exist and the data structures associated with it will be released.

9. THE OVERALL ORGANISATION OF MUSL

9.1. 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 language. 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 <<>,<>>, <!>, <[>, <]>.

9.1.1. 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.

9.1.2. 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.

9.1.3. 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 areas. 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 segments. 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 Chapter 23, but their function is not appropriate to understanding MUSL. 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 9.1.6 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.

9.1.4. 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.

9.1.5. 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 Section 2.5.2 and in Chapter 23.

9.1.6. 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.

However if the MUSL compiler is called with appropriate zeros in its MODE parameter it will call TL and TL.MODULE at the start and TL.END.MODULE and TL.END at the end. The least significant 8 bits of the compiler mode will be used as the TL mode.

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.

9.1.7. 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 9.5.11, 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 relevant detail a special import declaration (9.5.11) 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
```

9.1.8. 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, occurring 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 admissible. 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 entities declared in a block were actually declared in the enclosing procedure (or module) head, except that access to them is restricted to the statements of the block. This interpretation should be assumed in understanding the restriction stated below regarding the scope of variable.

When nested procedures occur in a module, the only variables 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 7, 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, WITHIN,
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 interpreted as a number.

They are represented by the '%' symbol followed by sequence of hexadecimal digits, and the notation allows multiple occurrences 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>!
                          *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 encoded itemised statement in hexadecimal,
        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. 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.

9.4.5. 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.

9.4.6. 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 relevant 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>:

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 forwards from the place where they appear. Labels may not be referenced in a '->' statement of an enclosed procedure, since a goto of this kind implies unwinding the procedure stack. However this restriction does not apply to label variables and parameters of type LABEL.

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, type LABEL, 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>/LABEL!<NAME>!
                  ADDR[<NUMERICTYPE>!<NAME>]!
                  ADDR<[>[<NUMERICTYPE>!<NAME>]<]>
                  ADDR<[>ADDR<]>!ADDR ADDR!ADDR
<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.

Variables of type LABEL may have labels (or other label variables) assigned to them. They may then be used as the operand of a 'goto' statement.

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.

When a type is preceded by ADDR, it denotes a pointer to an object of that type. ADDR may also be used as a numeric type to indicate an integer whose size is the address length of the target machine. Similarly a pointer to such an integer or vector of integer is denoted by ADDR ADDR, ADDR[ADDR]. This size may be specified in the mode parameter when the compiler is called (see 2.5.2).

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 should normally 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
```

If a user prefers to forfeit some degree of checking, the LSPECs may be omitted and the compiler will automatically generate LSPECs for any undefined procedure name which corresponds to a library procedure name in a currently open library. This option is controlled by a mode bit (section 2.5.2).

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        II=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>!<NIL>]!
                 TYPE<NAME> = <CONST>!
                 WITHIN<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.

It is sometimes necessary to use a TYPE name before giving its declaration. This may occur, for example, in importing procedures which use exported types. Thus the NIL alternative of a type declaration allows a name to be introduced as a type name. Its full definition must be given later in the usual way.

The last two alternatives of a type definition allow a type to be defined incrementally with several modules adding some fields. First of all a type name, and size in bytes, must be given using the construct

```
TYPE PRB.TYPE = 1000;
```

which states that the type PRB.TYPE will have a size not greater than 1000 bytes. The WITHIN statement which has the same syntax as a normal type definition, may then be used in any module into which the type is imported to add some fields. Only those fields declared in a module will be accessible in that module.

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 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 automatically declared name VSUB, providing it has not been redeclared, 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

```
<IMPORTS> ::= <IMPORT><IMPORTS>!<NIL>
 <IMPORT> ::= <PROC.DEC>!<VAR.DEC>!<IMPORT.DEC>!<TYPE.DEC>
```

In general a module may import any entity which is exported from another module. When the imported entity is a procedure or variable a full duplicate specification must be given before the module heading.

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>       ::= [LITERAL!VSTORE][<NUMERICTYPE>!<NIL>]!
                 TYPE!LABEL
<IMP.LIST>   ::= <NAME>[<[><]>!<NIL>][,<IMP.LIST>!<NIL>]
```

A type may be specified for an imported LITERAL or VSTORE and the NAME[] construct is used to indicate when a vector instance of VSTORE is to be imported.

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 is a label variable or parameter of type LABEL 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. Examples 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 CHARACTERISTICS OF THE COMPILER

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 statements in a manner described in Appendix C of the Fortran 77 standard. A Hollerith constant may only be used with INTEGER, LOGICAL or REAL types. The actual number of characters in such types is machine dependent (see 10.7).

The standard Fortran character set has been extended to include lower case letters, the tab and EOT characters. Lower case letters are accepted and treated as if they were upper case letters, everywhere except in character and hollerith constants. 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 third 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 beginning 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 implementation.

The compiler has a non-standard option of relaxing the strict checking between dummy and actual arguments of a subprogram. This option, when activated by setting a bit in the compiler mode parameter, permits an actual argument of REAL, INTEGER or LOGICAL type to be substituted for a dummy argument of either REAL, INTEGER or LOGICAL type, an actual argument of DOUBLE PRECISION or COMPLEX type to be substituted for a dummy argument of either DOUBLE PRECISION or COMPLEX type, and an actual argument which is an expression, variable, array element name or an array name to be substituted for a dummy argument which is a variable or an array.

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 may generate incomplete 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	Inconsistent 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.
130	Attempted use of unimplemented I/O feature.

Fortran also generates a trap Class 8 Reason 111 for an Assigned GOTO with a faulty argument specifying an invalid label. The mathematical functions library which can be called from a Fortran program also generates class 8 fault.

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 continued 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.

10.5.2. Import Library Procedures

The *IMPORT directive is provided to enable Fortran programs to call subprograms from a Library. The directive has as its arguments a list of procedure names represented by Fortran character constants or Fortran names.

*IMPORT 'CAPTION', 'OUTHEX', 'OPENFILE'

*IMPORT GFNIN, GFNOIUT are examples of Import directives. Note that as an imported procedure name may have more than six characters, a character constant is used. If the procedure names specified are not known to the system, then the Library containing them must be opened prior to the Fortran compilation. The procedures imported into a Fortran program can be called as if they were subprograms from the programs being compiled. This directive must occur before any Fortran statements.

10.5.3. Import Whole Library

The *LIB directive is provided to import all the procedures from a particular library. It is an equivalent to importing each of the procedures individually with a *IMPORT directive. The *LIB directive has one argument which is a Fortran character constant containing the name of the file containing the library. The named library must not be already open, as this directive will open the library.

```
*LIB 'TIMER/USR14'
```

is an example of the *LIB directive. This directive must occur before any Fortran statements.

10.5.4. Creating Fortran Libraries

The *EXPORT directive is provided to direct the compiler to include the named subprograms in the interface of a Fortran library. In addition to using this directive the mode parameter to the compiler needs the correct value (4) to create a library. The library will be placed in the file indicated by the second parameter to the Fortran compiler. The arguments to the *EXPORT directive are the Fortran names of the subprograms to be included in the library.

```
FORTRAN 0 LIB 4
      *EXPORT TEST
      SUBROUTINE TEST
      PRINT *,'TEST'
      END
      *END
```

is an example of creating a library called LIB containing the Fortran subroutine called TEST. The *EXPORT directive must occur before any Fortran statements.

10.5.5. Storage Control in Fortran

The *MAP directive is provided to give the Fortran programmer control of the location of code, common and local variables. This would be useful when several libraries share a common block, or if a program is very large. When this directive is not used Fortran handles the storage control automatically. The TL.SEG (23.4) procedure will need to be called in conjunction with this directive, to create and specify the MUTL segment in which to place the data or code.

There are three forms of the *MAP directive, for mapping code, common and local data.

The *MAP CODE form is used to specify the MUTL segment into which any following code is to be placed. This segment should already have been declared to MUTL by using the TL.SEG procedure. The TL.SEG may be called by using the ** command option. This directive has two arguments, the first specifies the MUTL segment number, the second (optional) parameter specifies the segment size in bytes.

```
FORTRAN  0  0  %8000
**TLSEG  3  %0 -1 -3 %C
             *MAP  CODE 3
              I = 1
```

is an example of the *MAP CODE directive which would place the code for 'I=1' (and any following code) into MUTL segment 3.

The *MAP CODE directive may occur anywhere within a Fortran program.

The *MAP for common is used to specify the MUTL segment into which the named common is to be placed. This segment should have already been declared to MUTL. The *MAP directive for common may only occur before a program unit, it cannot occur within a program unit.

```
FORTRAN  0  0  %8000
**TLSEG  3  %0   -1   -3   %C
**TLSEG  4  123456  -1  -3   %C
         *MAP  /NAME/  3
         *MAP  // 4, 123456
         SUBROUTINE ONE
         COMMON /NAME/ A,B,C
         COMMON D,E,F
         END
```

is an example of mapping the named common /NAME/ into MUTL segment 3 and blank common into segment 4 (of size 123456 bytes). If the common blocks are not mapped explicitly Fortran will choose store for them automatically.

The *MAP DATA directive is used to map local data variables and unmapped common (non blank) into a MUTL segment. The *MAP DATA directive has two arguments specifying the MUTL segment and its (optional) size in bytes. This directive may only occur before a program unit, it cannot occur within a program unit. A MUTL segment number of zero as an argument causes the explicit mapping to be disabled, and allows the Fortran to resume automatic store control of local data.

```
FORTRAN  0  0  %8000
**TLSEG  3  %0 -1  -3  %C
**TLSEG  4 %0  -1  -3  %C
            *MAP /COM/ 3
            *MAP  DATA 4
            SUBROUTINE TWO
            COMMON /COM/ A,B,C
            COMMON /OTHER/ D,E,F
            INTEGER I,J,K
            I = J+K
               .
               .
               .
            END
```

is an example of mapping the common /COM/ onto MUTL segment 3, and the common /OTHER/ and the variable I,J,K onto MUTL segment 4. If the *MAP DATA is not used Fortran will allocate storage space automatically.

The use of mode bit 8 of the compiler also affects storage management. The setting of mode bit 8 to place items on the stack takes precedence over the *MAP DATA directive.

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).

The details of pre–connection are left undefined by the standard. In the implementation 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 (unless it's an I.O stream), and then it is unambiguous 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.

It is not possible, at present, to BACKSPACE unformatted records, neither does the implementation allow the changing from reading to writing on pre–connected streams which are only defined for one mode of operation (an I.O stream should be used instead).

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 two parameters (the file to be printed, and the destination device), 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.

10.7. IMPLEMENTATION DEPENDENCE

10.7.1. Accuracy

10.7.1.1. 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.

INTEGER, LOGICAL and REAL may hold up to four characters.

10.7.1.2. VAX

On VAX INTEGER is in the range –2**31 to 2**31, REAL has a mantissa of 24 binary digits (approximately 7 decimal digits) with an exponent range 2**64 to 2**(–64) which gives a maximum decimal magnitude of 10**19 approximately.

DOUBLE PRECISION has an accuracy of 16 decimal digits approximately.

INTEGER, LOGICAL and REAL items may hold up to four characters.

10.7.1.3. MC68000

On MC68000 INTEGER is in the range –2**15 to 2**15. REAL has a mantissa of 24 binary digits (approximately 7 decimal digits) with an exponent range of 2**64 to 2**(–64) which gives a maximum decimal magnitude of 10**19 approximately.

DOUBLE PRECISION has the same accuracy as REAL.

INTEGER and LOGICAL may hold up to two characters and REAL up to four characters.

The action of the EQUIVALENCE is non–standard. INTEGER and LOGICAL are allocated 2 bytes, REAL and DOUBLE PRECISION 4 bytes, and COMPLEX 8 bytes.

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.

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 8 of the compilation mode is used to permit any non–initialised local data to be placed on the stack.

Bit 14 of the compilation mode is used to suppress the type checking between actual and dummy arguments of a subprogram.

Bit 13 of the compilation mode is used to indicate that the Fortran statements are to be listed as they are compiled. This feature may be useful for debugging purposes. Comments are not listed.

Bit 15 of the compilation mode is used to permit the interpretation of imbedded MUSS commands occurring during compilation.

11. COBOL

11.1. INTRODUCTION

The MUSS COBOL compiler will implement the majority of the high subset of COBOL as defined in the document 'Draft proposed American National Standard Programming Language COBOL BSR X3.23-198X'. The initial implementation will consist of the following functional modules:

Nucleus	Level 2
Sequential I-O	Level 2
Relative I-O	Level 2
Indexed I-O	Level 2
Inter-Program Communication	Level 2

All input/output will operate on files and documents as provided by the MUSS library, no direct handling of files on magnetic tapes and discs will be provided initially.

12. PASCAL

12.1. INTRODUCTION

The Standard Pascal referred to here is as specified by the working draft of ISO. ISO/TC 97/SC 5 N 595, January 1981. In general the full standard language is implemented but some implementation restrictions and extensions are listed below.

12.2. RESTRICTIONS

1. Character strings are restricted to a maximum length of 140 characters.

2. Sets are restricted to 128 elements. Set types may be defined over any base type which is:-

   (a) an enumerated type (no element beyond the 128th. being used), or
   (b) a type integer (or a subrange of) with a minimum element >=0 and a maximum element <= 127, or
   (c) one of the types char or boolean.

3. Run-time checking is restricted to detection of the following errors:

   (i) Subrange variable out of range.
   (ii) Non-existent CASE chosen.
   (iii) Illegal pointer access (only in simple cases).

12.3. EXTENSIONS

With the I+ option all deviations from the ISO standard are monitored but code is still compiled for these extensions whenever possible, with I- the following extensions are permitted.

1. Facilities are available to call any suitable MUSS library procedure. To conform to the ISO standard any such procedure used should be preceded by a procedure-declaration or function-declaration with an external directive. The parameter specification given will be checked for compatibility with the MUSS specification. However the compiler allows such procedures to be called without declaration except when the procedure has a

user–defined type parameter, when an external declaration must be given. Additionally the following functions are provided

SIGN(X1,X2)
DIM(X1,X2)
LOG10(X)
ARCSIN(X)
ARCCOS(X)
SINH(X)
COSH(X)
TANH(X)
SIGN(X)

where Xs are either integer or real type. These each correspond to one or more of the MUTL mathematical functions.

2. An OTHERWISE option is provided for case statements where the case constants cannot all be listed. If the CASE expression is of integer (i.e. not subrange) type then an error will be registered rather than the OTHERWISE alternative being chosen if the expression is outside the range: LOW .. HIGH where LOW is the minimum of −1000 and the lowest CASE constant listed and HIGH is the maximum of +1000, and the highest CASE constant listed. Thus a wider range can be obtained by suitably choosing the CASE constants listed.

3. Hexadecimal constants are accepted. The hexadecimal digits must be preceded by % and be at most 32 bits.

4. Any line commencing ** is interpreted as a MUSS command to be obeyed at compile time.
The main use of this feature is to include source text from other files, or to direct the action of the MUTL system when compiling a library (see 12.8) or placing variables in a specific segment (see reference 12.9).

5. Renaming of files is permitted in the program parameters. (See 12.5)

12.4. CONFORMANCE TO PASCAL TEST SUITE

The compiler has been validated against the Pascal processor validation suite of B.A. Wickman and A.H.J. Sale. Details of the results are in the PASCAL Implementation Manual Section 2.4.

12.5. INPUT AND OUTPUT

The files INPUT and OUTPUT are predefined to be the current MUSS input and output stream when the program run is started or the library opened.

All external files must be specified as parameters in the program heading corresponding to the MUSS files of the same name. An extension allows parameters of the form Pascal filename = MUSS document where MUSS document is a sequence of characters terminated by , or) as described in Chapter 3.

There is no limit as to the number of files a programmer may use but the maximum number of "readable" files is 8 and that of "writable" too is 8. All files must be initialized to "readable" and "writable" by the procedures RESET and REWRITE. If a programmer wishes to read from a file previously written on, he only has to call the procedure RESET and vice versa. If the number of "readable"/"writable" files is insufficient then the procedure CLOSE(f) may be called to reduce the number of files in that state.

The correspondence between Pascal and MUSS files is as follows:

(i)	MUSS sequence of characters is Pascal TEXT or FILE of CHAR.
(ii)	MUSS unstructured sequence of binary is Pascal file of basic type.
(iii)	MUSS sequence of units is Pascal file of records without variants.
(iv)	MUSS sequence of records is Pascal file of records with variants.

The Pascal input/output system imposes an additional level on the basic MUSS system, so extreme caution is needed if basic MUSS I/O commands are mixed with the Pascal facilities. At compile time the compiler uses the basic MUSS facility.

Note: To avoid reading beyond the end of a file, all input files must be terminated by a newline character.

12.6. COMPILE TIME PRAGMATS

These are of the form (* $<letter> + *) to turn on feature <letter> and (* $<letter> – *) to turn this feature off.

The allowed letters are:

D <integer>	Direct variables declared beyond this point into new code area.
C	Compile full run-time checking (default –).
I	Check conformance to ISO standard (default +).
E	Turn on error listing (default +).
L	List source text (default –).
P <integer>	Call TL.PRINT (<integer>).
S <integer>	Direct compilation into new code area.
T	Compile run-time line number trace (default–).
Y	Following procedures are to be exported library procedures, when compiling library (default–).
R	Make all real variables beyond this point double precision (default–).
N	Make integer size 16-bit (default– i.e. 32-bit). N.B. This may only be used at the start of compilation.

12.7. COMPILE TIME ERRORS

Error messages are printed below the faulty source line with arrows indicating the symbol which was faulted. Arrows separated by commas indicate more than one error on the same symbol.

12.8. PASCAL LIBRARIES

MUSS Pascal does not support a full module facility but it is possible to compile a set of Pascal procedures into a library.

To compile a library the Pascal compiler is called with third parameter = 4 (see 2.5.2). The user has control over which procedures are in the external library interface. The Y+ pragmat must precede the first such procedure heading. If subsequent procedure are not required Y– is used to inhibit addition to the library interface.

If the library has any variable declarations at the program level these must be identical to those of the calling program as imported or exported variables are not implemented.

Example of a Pascal library consisting of two procedures Q and S.

```
  **TLSEG 0 0 %40000 -1 6     (Allocate Vax segment 4 as runtime address of
library)
  PROGRAM P;
  (*$Y+*)                      (Put Q in library interface)
  PROCEDURE Q(P1 : INTEGER);
  BEGIN
  WRITELN('PROC Q : P1 = ',P1)
  END;
  (*$Y-*)                      (R not to be put in library interface)
  PROCEDURE R(C : CHAR);
  BEGIN
  WRITE(C)
  END;
  (*$Y+*)                      (Put S in library interface)
  PROCEDURE S(VAR C : CHAR);
  BEGIN
  WRITE('PROC S : C = ');
  R(C);
  END;
  BEGIN
  END.
```

12.9. RUN TIME STORE USE

The Pascal stack segment for local variables is the MUSS stack segment. (This is one full segment on the VAX).

The Pascal heap segment is currently segment %21, on the VAX. If more space is needed it is allocated dynamically. Heap space is recovered by dispose and reused. A procedure ATTACH (SEG.NO, PTR) is available in the run-time library to set a pointer to the start of a given segment. A warning message should be expected when this procedure is called from a Pascal program because the type of the second parameter will not correspond to the specification given for it. In order to use this procedure the following declaration should be made in the calling program:

```
  PROCEDURE ATTACH (P1:INTEGER; VAR P2:<required pointer type>);
                                                          EXTERNAL;
```

The present implementation creates full size records for variant records without attempting to optimise the space allocated. The layout of fields within records is controlled by MUTL, presently to the byte level. Packed does not, at present, attempt any further packing to the bit level.

Pascal uses MUTL to create space for variables in the current data area. Variable declarations can be directed into other segments of store using **TL.SEG, **TLLOAD, and **TLDATAAREA as described in 23.4.

```
PROGRAM P;
e.g.      **TLSEG 2 0 %40000 -2 12
          **TLLOAD 2 3
          **TLDATAAREA 3
          VAR X,Y,Z: INTEGER;
          **TLDATAAREA 0
          a,b,c: real;
```

puts the variables X, Y, Z in a segment 4 (on a VAX) and a, b, c on the stack. Note that Pascal uses TLSEG 1 for the heap so this may not be reused. When cross-compiling the pragmat (*$D<integer>*) should be used instead of **TLDATAAREA to avoid calling the wrong MUTL.

13. VIRTUAL STORE MANAGEMENT

Each process in MUSS is provided with its own virtual store. The virtual store is assumed to be segmented, and a set of procedures are provided to enable a process to create/release segments and change their attributes. The exact number and limiting sizes of the segments varies between MUSS installations, and not all of the segments may be directly accessible. For this reason, two types of segment are distinguished.

The first type is for areas which must be randomly accessible, such as for code or stack/buffer areas. In general, these will be demand loaded by the operating system, and may be accessed by generating an appropriate machine address. On some machines, particularly small (16 bit) machines, the actual addressing capability of the hardware is too restrictive. A process is therefore allowed a larger number of segments than can be addressed directly, and facilities are provided to allow a user to MAP a subset of the segments into the addressable space. (The exact number is machine dependent). The MAP function is also used on machines which are unable to support demand loading, to inform the system which segments must be in store whilst the process is running.

The second type of segment is suitable for uses in which there is a well defined access pattern, such as for input/output documents. This type of segment is known as an X.SEGMENT, and is characterised mainly by the fact that its maximum size is not determined by the physical restrictions of the central processor. For example, on the PDP11, an X.segment may be created which is very much larger than the addressable space of a process (64 Kbytes). As these segments cannot be addressed directly, the only way in which they may be accessed is by copying individual blocks into/out of accessible segments. A command, COPY.BLOCK, is provided for this purpose. In all other respects, the two types of segment are treated in a similar way, and the commands described in this chapter to enable a process to manipulate its virtual store, will operate on either type of segment. Also either type of segment may be sent as a message or filed.

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 current 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 write access in user mode, with authorisation to change the permission.

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. If P2 is negative or zero, a single segment of maximum size will be created.

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) CREATE.X.SEGMENT(I)

This procedure creates an X.segment on behalf of the current process. Parameter P1 is the segment size in blocks (block size = 1024 bytes on PDP11).

The procedure returns two values. A segment identifier is returned in PW1, and this should be used in all subsequent commands when referring to the segment. The size allocated for the segment (in blocks) is returned in PW2. As for create segment, this may differ from P1 if rounding has taken place.

(3) 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).

(4) CHANGE.SIZE (I,I)

This procedure changes the size of the segment specified in P1 to the size given by P2. If the segment is an X.segment, P2 is the new size in blocks, otherwise it is the size in bytes. As with the create segment command the size may again be rounded by the system. 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 not 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.

(5) 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.

(6) INTERCHANGE (I,I)

This procedure interchanges the specification of the two segments given by the parameters. It is permissible for either (or even both) of the segments to be undefined, and in this case the effect is simply to redefine the existing segment.

(7) READ.SEGMENT.STATUS(I)

This procedure reads the status of the segment specified, and returns in PW1 and PW2 the size in bytes (blocks for an X segment) 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 not directly accessible
```

(8) MAP(I,I,I)

This procedure is used primarily on machines with small or otherwise unusual virtual stores to ensure that a segment is accessible. P3 specifies the number of segments to be made accessible, starting at segment number P1. P2 indicates how the segment may be mapped in the virtual store.

On machines which must have the segments locked in store before they can be addressed (such as MC68000), this procedure will ensure that the segments are always in store when the process is running. In general, a limited number of segments may be locked in at any one time. P2 identifies a position within this subset of segments. If negative, a free position will be allocated. The same P2 (with P1=0) should also be specified when the segment is to be unlocked.

On machines with small virtual stores (e.g. PDP11), this procedure maps the segment into part of the addressable space. In this case, P2 specifies a segment position within the addressable space. It should also be used, again with P1 = 0, when a segment is to be removed completely from the addressable space.

On machines, such as VAX, with a large virtual store which can be demand paged, this procedure has no effect other than setting the PW results.

The procedure sets PW1 to the number of the segment which previously occupied that area of the virtual store. PW6 is set to the segment number to be used in address calculations to access the segment. This procedure has no effect if an X.segment is specified in P1.

(9) 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.

(10) CREATE.CS(I,I)

This procedure takes the segment specified by P1 and causes it to become the common segment specified by P2. It applies to all processes and it lasts until the system is re–booted. There are a limited number of common segments which the command may specify and they are determined by the local configuration state of each machine.

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 register of the process will be saved on the stack before calling the procedure. Restoring these will 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 units.

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)
1 spare byte
Suspension reason (1 byte)
Status (1 byte)
Priority (1 byte)
CPU time used
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 CPU units, 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. The time remaining in the local CPU time is returned in PW3.

11) 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 if 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.

15. INTER-PROCESS COMMUNICATIONS

Processes in MUSS normally communicate with one another by sending messages, and the input/output devices connected to the system also generate and receive 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. Specially encoded SPNs are used to address the peripheral devices. The message system is designed to be network wide, and so part of the SPN identifies the machine containing the destination process or device.

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.

An alternative to the message system is provided for those situations where buffered input/output is unsatisfactory, for example, in a word processing package. Direct single character communication with a device is provided as described in 15.2.

15.1. MESSAGE 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 on values as shown in Table 15-1.

Table 15-1. Mode Encoding

Value	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-%F	A reply may be sent on a channel number equal to the bottom 3 bits (i.e. channels 0 to 7).	The bottom 3 bits specify the 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 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 or zero, a free segment number is allocated. PW1 always contains the segment number actually used (zero 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.

15.2. DIRECT COMMUNICATION COMMANDS

These commands provide an alternative more direct way of communicating with peripherals which are logged into a message channel.

1) CHANGE.CHANNEL (I,I,I)

This procedure must be called with P2 = 0 before any attempt is made to communicate directly with a peripheral. P1 specifies the SPN of the required peripheral. This SPN will be included in the source information supplied when reading a message from the peripheral. It can also be obtained by calling I.SOURCE for any message stream which has received input from the peripheral. The procedure informs the input/output manager of a change in the mode of usage of the device and returns in PW1 an identifier which must be used as the P1 of the following three commands. No further messages will be sent to the process whilst the device remains in this state, but output messages may still be sent from the process. The procedure is also used (with P2 = 1) to change a direct communication channel back to a message channel.

P3 is a parameter giving the required device modes of the input and output devices associated with the channel. The least significant eight bits contain the input device mode and the next eight the output device mode. The significance of the bits is described in Chapter 20. The previous modes of the devices are returned in PW2.

2) READ.CH (I)

This procedure reads the next character of input from the device specified by P1. If there is no input waiting in the buffer the called process will be suspended until the next input character arrives.

3) WRITE.CH (I,I)

This procedure transmits the character specified by P2 to the device specified by P1. The process may be halted if, for example, previously sent messages have not yet been transmitted or if too many characters are buffered awaiting transmission.

16. CONVENTIONS FOR NETWORK COMMUNICATIONS

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 4 bytes 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 4 bytes (or byte in a short message) contain an integer count of the number of bytes within the data. Each of these 32 bit integers is given in the form least significant byte first, most significant byte last. The size of the preamble is not included in the data 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.

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 20.

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,[C])

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,I)

This procedure skips over a number of blocks of data on the specified device. P1 is the unit number and P2 the direction to be travelled (0 = forwards, 1 = backwards). P3 gives the number of data blocks to be skipped. At least 1 block will always be skipped, even if P3 is zero. 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,I)

This procedure skips across data blocks on a medium until a tape mark is encountered. P1 specifies the unit number of the tape and P2 the direction of travel (0 = forwards, 1 = backwards). P3 specifies the number of tape marks to be skipped.

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 formats determine

i) whether labels are present on the tape
ii) whether MUSS housekeeping information (as described in Chapter 16) is also stored with the file.

Two conventions for labelling tapes are currently used. The first is according to ANSI standard X3.27, which defines the sequence for volume labels, file labels and tape marks. The second is where labels are omitted, and the only textual information on the tape is that of the actual files. A single tape mark separating each file is the only control information used in this case. For both formats, a double tape mark signifies end of tape.

The utilities for reading/creating tapes have a parameter to specify the format required. This parameter may take the following values:

0	Files include MUSS housekeeping information, ANSI standard labels are written to the tape. This is the normal form for transmission of files between MUSS systems.
1	Housekeeping information is removed from the files. ANSI standard labels are included.
2	No labels are written to the tape. Files include MUSS housekeeping information.
3	No labels or housekeeping information are written to the tape. This format is often used for transmission of textual files between incompatible systems.

1) WRITE.FILE(I,[C],I,I,[C],[C],[C],I)

This procedure writes a file P2 to the tape on unit P1. P3 specifies the format to be used for the file. The size of the data blocks on the tape is given by P4. If this is negative or zero, a default size of 1 Kbyte will be used.

Parameters P5 and P8 are only applicable to tapes written in ANSI format, and specify information to be inserted into the header label which precedes the file on the tape. P5 is the sequence number of the file on the tape; P6 is the file generation number; P7 is an expiry date for the file and P8 is an accessibility key. The expiry date is specified as five digits, the first two giving the year and the next three the day within the year. Suitable defaults are supplied for P5 to P8 if specific values are not given.

2) WRITE.TAPE(II,[C],I,I,I)

This procedure writes a set of files to a tape. The parameters P1 and P2 specify a tape name and tape label respectively. P3 states the format to be used for the individual files on the tape. If ANSI standard format is chosen, a suitable volume label will be written at the start of the tape before the individual files. P4 specifies a block size to be used for the files, with a default as for WRITE.FILE. P5 indicates whether a number of files already exist on the tape. Thus, if P5 is non zero, the procedure skips to the current end of the tape before appending the new files.

The procedure prompts for the names of the files to be written. If a ? appears in a filename, the procedure selects a set of files from within the current directory whose names commence with the string immediately preceding the ?. If a ? on its own is typed, the entire directory will be copied to tape. A file name of @ indicates that the tape is complete, and appropriate tape marks are written to the tape.

3) READ.FILE(I,[C],I,[C])

This procedure reads a file from the tape mounted on unit P1. P2 gives the filename which will be used for securing the file. P3 specifies the format of the tape. If ANSI format is used and a null filename is specified, a filename will be extracted from the file header label on the tape. P4 is only applicable to ANSI tapes, and should be a vector into which the 80 character file header label is returned.

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. P3 states the format of the tape. If the tape is in ANSI format, the names of the individual files are extracted from the file header labels preceding each file on the tape. For simple textual tapes, the procedure prompts for the individual file names before reading each file. The procedure returns when a double tape mark is encountered.

5) FIND.FILE(I,[C])

This procedure searches an ANSI format tape for a file with name P2. P1 gives the unit number of the tape drive.

At the end of the command, the tape is in a position where a subsequent call of READ.FILE will copy the file from the tape. As the search is in a forward direction along the tape starting at the current position, a file will not be found if it has already been passed over. If the user does not know the relative position of the required file on the tape, an MT.REWIND command should be issued initially to ensure that the complete tape is searched.

6) 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, and 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, but an attempt is made to continue the copying process.

18. BENCHMARKING AND PERFORMANCE MEASUREMENT

This Chapter describes the procedures provided in MUSS for interactive benchmarking and collecting statistics about the system.

18.1. COMMANDS

There are two categories of commands associated with benchmarking and performance measurement, namely an organisational function for acquiring system statistics and a number of high–level procedures for printing statistics and benchmarking.

18.1.1. Organisational Command

1) SYSTEM.STATS()

This procedure returns (in PW1) a new segment, containing statistics about the system since restarting.

The following integers will be returned in the segment:

Number of organisational commands called
Time at user level
Time at command level
Time at interrupt level
Time at appendix level
Number of drum transfers in
Number of drum transfers out
Total number of blocks transferred
Elapsed time (seconds)
Number of processes created
Number of segments created
Number of X segments created
Number of long messages sent
Number of short messages sent
Number of files opened
Number of files created

Number of files updated
Total number of tasks set

18.1.2. High-Level Procedures

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.

3) INT.JOB.JOB([C],[C],I)

This procedure implements an interactive benchmarking facility. It runs a set of interactive jobs, supplied as input 'scripts' on files. Each script is run from a specified number of terminals, and may if required be run more than once from each terminal. The scripts should not contain the command STOP.

Input is supplied to the jobs a line at a time, each line being sent only after receiving a prompt in response to the preceding line. Prompts are recognised by the character NULL, which appears in all standard system prompts.

On completion of all scripts, the system statistics collected during the session are output to a log. In addition a complete record of all short messages input and output is kept in the log.

P1 is name of the supervisor used for job initiation (default is JOB). P2 is name of the output document (log) on which the session log is to be output. If P2 is not specified, the currently selected output stream will be used. P3 is a numeric delimiter character, used to parameterise the scripts. If this is non-zero, then any occurrence of this character within the scripts will be replaced by a numeric terminal identifier. As a result several terminals may perform similar scripts operating upon different files.

Delays can be put in the scripts by inserting lines which start with " ¯ " followed by either an integer, specifying the delay in seconds, or by the delimiter character, which generates random delays.

A list of the files containing the scripts, with the number of terminals and number of repetitions for each script, should appear on the current input stream, terminated by the character '@'.

Example:

```
***A JOB USER PASS IJJ C1000
INT.JOB.JOB(JOB,LOG.FILE,?)
SCRIPT1 20 5
SCRIPT2 5 15
@STOP
***Z
```

If INT.JOB.JOB is initiated interactively, appropriate prompts will steer users.

19. ACCOUNTING AND USER MANAGEMENT

Controlling users and accounting for their use of the system is dependent on 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 four resources. These are:

(i) CPUTIME	The amount of CPU time used (time slices).
(ii) NO OF FILES	The maximum number of files that a user can have in his directory.
(iii) FILESTORE	The maximum amount of filestore (Kbytes) a user can occupy.
(iv) SUBORDS	The maximum number of subordinates a user can have.

The user, SYSTEM has infinite amounts of all resources which it can distribute to other users in the system. Any other user can create new subordinate users, if it has an allocation for SUBORDS.

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 commands associated with accounting and user management fall into two categories, basic (primitive) commands and high-level commands. The primitive commands are those which deal with separate accounting and user management tasks, while the latter commands are only a collection of applications for primitive ones.

19.1.1. Primitive Commands

The set of consumable balances and incomes and reusable maxima for a user is called his "parameters". The parameter numbers for the standard resources are:

CPU time balance	- 1
CPU time income	- 3
Maximum files	- 4
Maximum file store	- 5
Maximum subords	- 6

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)

This procedure removes current user's subordinate (with username P1) from the system. The subordinate's balances of consumables and maxima of reusables are added to the current user's corresponding values. The subordinate must not have any subordinates or used any reusables.

3) SET.USER.PARAM(II,I,I)

This procedure re-distributes a balance, maximum or income between a user and one of his subordinates (P1). P2 is the parameter number. P3 is the amount (POSITIVE) to which the parameter is to be set. The change must not reduce either user's parameters to less than zero.

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)         |
| LEVEL (1 byte) *           |
| spare (3 bytes) *          |
| BALANCE 1 (4 bytes)        |
|                            |

|____________________________|
| BALANCE m (4 bytes)        |
| INCOME 1 (4 bytes)         |
|                            |

|____________________________|
| INCOME m (4 bytes)         |
| MAXIMUM 1 (4 bytes)        |
|                            |

| MAXIMUM n (4 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 reusable 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 an authorised user.

7) FIND.U(II,II)

This procedure returns in PW1 the UID of the user specified by username (P1) and password (P2). The UID of the subordinate may be obtained without giving a password.

19.2. HIGH-LEVEL COMMANDS

1) NEW.USER([C], [C])

This procedure creates a new set of users as subordinates of the current user.

Prior to the creation of individual users, values for new users resources are read from P1. This should contain CPU time income, maximum files, maximum file store (1/2K blocks) and maximum subordinates. Entries are all unsigned decimal integers on separate lines. An empty line or one which has other than integer value in it is interpreted as a zero. This list may be terminated using an "@" on a separate line, in which case the rest of the resources will be set to zero.

Names of users are read from another document (P2). This must contain pairs of username and password, each on a separate line. The default for password is "X", however if more users are to be created the default password must also appear in P2. Operation may be terminated by an "@".

Default for P1 and P2 is the currently selected stream. When the command is used interactively, appropriate prompts will appear to steer the operation.

2) DELETE.USER([C])

This procedure removes a user's subordinate from the system, transferring his resource allocations back to his superior. The subordinate must not have any subordinates,or have used any other reusables. P1 is the source document which contains names of users to be deleted, terminated by an "@". The default is the current stream.

3) SET.ACCOUNT([C],[C],[C])

This command redistributes balance, income and maxima between a user and one or more of his subordinates. It works in a similar way to the command NEW.USER by reading new income/maxima, new balance (CPU time) and finally usernames.

P1 is the source from which positive integers indicating the new values of the user's income/maxima are to be read.

P2 is the source from which balance is read. Both P1 and P2 may be terminated by an "@" on a separate line. In each case the remaining resources will be left intact.

Usernames will be read from P3, one name per line and terminated by an "@". The default for P1, P2 and P3 is the currently selected stream.

4) LIST.ACCOUNTS(C,[C])

This command lists the accounts parameters for the current user and his immediate subordinates (P1 equals S), or all his subordinates (P1 equals A).

For each user, the name and level (0 = self, 1 = subordinate, etc) and a set of accounts with appropriate captions are output to the destination document (P2). The currently selected output stream is the default for P2.

20. OPERATOR FACILITIES

20.1. CONFIGURATION

The operator is provided with facilities for changing the configuration of a machine from that initially compiled into the system. The changes may affect both the peripheral process configuration of a machine, its status within a network, and the detailed device configuration.

Two options are available to an operator for changing the configuration. A set of commands, as described below, may be called to change the configuration whilst the system is running. These changes will be lost when the system is restarted. For changes of a more permanent nature, a text file may be created containing a set of configuration parameters. This file will be processed and the configuration amended accordingly on every system restart. The file must be called CONFIG and exist under the SYSTEM username.

It is necessary to understand some of the terminology used in describing the organisation of input/output devices before attempting to alter the configuration of the system.

First each 'device' attached to the system has a 'logical device number' which is assigned to the device when it is configured (see the CONFIG command below). A device in this sense is any input or output device, thus a VDU represents two devices, one of each.

Some output devices may also have 'peripheral channel numbers' assigned to them, again by the CONFIG command. These numbers serve two functions. They are the means by which an input device is 'linked' with an output device. The implication of this is that all replies sent to the input will appear in the linked output including the monitoring of logging in errors. The norm is for the two devices comprising a VDU to be linked. Another use of the peripheral channel number is in forming a 'SPN' for a peripheral process. This is necessary, for example, in the SET.PERIPHERAL command, which allows a pseudo process name to be assigned to devices such as LPT. Each device on the system has a unique 'physical device identifier' which has the following format:

```
 _____________________
|    8    |    8    |
 ---------------------
     |         |
     |          ---------- physical device number
     |
      -------------------- physical device type
```

The physical device type field indicates the actual kind of the device, and distinguishes between devices which have major differences in their handling. The device number gives the internal device/line numbering within each type.

A system process number (SPN) in general is formed from a process number and a machine number. However, one bit is reserved to indicate that the process number field is to be interpreted as a peripheral channel number thus

```
 ________________________________
|                |1|    8       |
 ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
         |        |     |
         |        |     -- machine number
         |        ------ 1 = peripheral, 0 = process
         ------------ process number/peripheral channel
```

20.1.1. Configuration Commands

When the system is in operation the communication between users and supervisors and user processes and peripherals is determined by the contents of the following tables

NETWORK TABLE -	this gives the name of each machine accesible from the current machine, and the SPN for the peripheral channel through which it can be accessed.
SUPERVISOR TABLE -	this gives for each supervisor in the network - a name, a machine name, a SPN, a PID and a channel.
PERIPHERAL TABLE -	this gives a name and a SPN for each peripheral process.

The first three commands below define the entries in these tables.

1) SET.NETWORK(I,II,I)

This procedure changes entries in the NETWORK TABLE. The first parameter is the table entry to be changed. P2 specifies a machine name, and P3 a peripheral SPN which will be used for communication between the current and specified machine.

2) SET.SUPERVISOR(I,II,II,I,I,I)

This procedure changes entries in the SUPERVISOR TABLE. The first parameter is the table entry to be changed. P2 and P3 specify a process name and machine name respectively for the supervisor process, and P4, P5 and P6 are the SPN, PID and channel for communication with the process.

3) SET.PERIPHERAL(I,II,I)

This procedure changes entries in the PERIPHERAL TABLE. The first parameter is the table entry to be changed. P2 specifies a peripheral name and P3 the SPN to be associated with it.

4) REROUTE.PERI(I,I,I,I,I)

This procedure allows the operator to temporarily change the routing of documents to a peripheral. The first parameter specifies the table entry to be chaged. P2 is the peripheral SPN whose documents are to be rerouted. P3, P4 and P5 specify an alternative destination for the documents, as a SPN, PID and channel number.

5) CONFIG(I,I,I,I,I)

This procedure is used to change the configuration of devices in the system.

Its parameters are

P1 – Logical device number
~~P2 – Physical device identifier (see above)~~
P3 – Physical number of the paired device
P4 – Output channel number
P5 – Logical mode
P6 – Driver mode
P7 – Line Parameters.

The logical device numbers (P1) are arbitrary assigned providing the maximum configured number of devices is not exceeded. Thereafter the logical device number is the means of identification for device reconfiguration.

Devices may be paired (P3), which means they are treated as one from the point of view of logging in and logging out. Normally the screen and keyboard of a VDU are treated as a pair and unidirectional devices like printers and card readers are not paired at all.

Output channel numbers (P4) are assigned to outputs so that inputs may be linked with them and sometimes names assigned to them. If an output channel number is given whilst configuring and input device it is 'linked' to that output.

The logical mode (P5) has the format

```
 ____________________________________________
|      8      | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |   0            1
              |
       |               |   |   |   |   |
       |               |   |   |   |    -- input       / output
       |               |   |   |    ---- single        / paired
       |               |   |    ------ shareable       / dedicated
       |               |    -------- short             / long message
       |                ---------- spooled             / interactive
       |
        ---- logical device class
```

The logical device class distinguishes between character and communications protocols and there may be variations in these between different installations although the first six classes will be constant thus

0	character input
1	character output
2	MUSS communications input
3	MUSS communications output
4	talk input
5	talk output

The functions of the least significant 8 bits of the logical device mode, thought of as logical device type, should be mainly self explanatory, but the following notes may help.

paired	implies the device is one of an input–output pair (such as an interactive terminal) and so any logging out operation should apply to both members of the pair.
dedicated	(output devices only) means the device must be dedicated to one process at a time.
long messages	means that long messages can be input/output via this device. For very slow devices, the use of long messages could be expensive.
interactive	(character devices only) distinguishes between batch and interactive devices. Lineprinters, and paper tape devices are, for example, considered as batch devices, while VDUs are interactive devices.

The 'driver–mode' controls the special actions of the device driver software according to the following bit allocations

Input device:

```
 _______________________
|1|1|1|1|1|1|1|1|          0 = OFF        1 = ON
 | | | | | | | |
 | | | | | | | |
 | | | | | | |  ---- Xfer on LF, CR, FF
 | | | | | |  ------ Ignore ETX and transfer
 | | | | |  -------- Xfer on all
 | | | |  ---------- Ignore STX and pre-amble
 | | |  ------------ CR ->LF
 | |  -------------- BS + DEL
 |  ---------------- ETX = BREAK
  ------------------ Note XON/XOFF
```

Output device:

```
 ___________________
|  |1|1|  |  |  |1|   0 = OFF   1 = ON
    | |           |
    | |           |
    | |           -- BREAK is ENGAGE/DISENGAGE
    | |
    | |
    |  ------------ LF -> CR+LF
     -------------- FF produces DISENGAGE
```

The line parameters (P7) specify the physical characteristics of the device as follows

```
 __________________________
|    |    |    4    |2|2|2|
               |     | | |
               |     | |  -- parity (0=even, 1=odd, 2=mark, 3=space)
               |     |  ---- no. of bits (0=five, 1=six, 2=seven, 3=eight)
               |      ------ no. of stop bits (0=1 stop, 1=1.1/2 stops,
               |             2=2 stops)
                ----------- transfer rate (see below)
```

The following encoding may be used to specify different transfer rates:

	transfer	rate
code	baud	character/s (approx.)
0000	110	10
0001	300	30
0010	1200	120
0011	2400	240
0100	4800	480
0101	9600	1K
0110	19200	2K
0111	38400	4K
1000	76800	8K
1001	153600	16K
1010	307200	32K
1011	614400	64K
1100	1228800	128K
1101	2457600	256K
1110	4915200	512K
1111	9830400	1M

The maximum number of devices and peripheral channels which may be configured are determined at compile time and care must be taken to keep P1 and P2 within these limits.

6) CONFIG.ENQ(I)

This procedure provides the means for enquiring about the configuration of a particular device, with P1 giving the physical identification of that device. If P1 is −1, the system scans its device table and returns the information about the first device which is used by the calling process. The actual configuration attributes are returned in PW1, PW2, PW3 and PW4. PW1 will contain the logical device number which should be used on a subsequence call to the CONFIG command. The logical device identifier is returned in PW2, while the physical device identifier and the physical device mode are returned in PW3 and PW4, respectively. The formats of these attributes are identical to those described for the CONFIG command.

20.1.2. Configuration File

The CONFIG file consists of a set of keywords identifying the type of parameter to change, followed by the changes. The keywords are preceded by @. The following options are available

@MACHINE	This should be followed by an integer specifying the number of the machine within the network.
@NETWORK	This allows the operator to change the configuration of the machines known to the system. The keyword is followed by a list of machine names and associated peripheral SPNs. These correspond to the parameters for the SET.NETWORK command below.
@SUPERVISOR	This allows the operator to change the list of supervisor processes known to the system. The keyword is followed by a list corresponding to the parameters of the SET.SUPERVISOR command, of process and machine names for the known supervisors, and their associated SPN, PID and channels.
@PERIPHERAL	This allows the operator to change the list of peripheral names wired into the system. The keyword is followed by a list corresponding to the parameters of SET.PERIPHERAL, of peripheral names and their associated peripheral SPN's.
@ROUTE	This allows the operator to change the routing information for a peripheral message channel, thus allowing output for a peripheral to be redirected to an alternative device. The keyword is followed by a list of peripheral SPN's and alternative SPNs, PIDs and channels for the destination.
@DEVICE	This allows the operator to set/change device configuration. It has a parameter list which corresponds to that of the CONFIG command.
@END	End of parameters.

All numbers appearing in the file must be hexadecimal.

A typical CONFIG file for configuring a machine might be

@MACHINE 3	This is machine number 3 in the network.
@NETWORK VAX01 103 VAX02 303	Two other machines are known, and these can be addressed via peripheral SPN's %103 and %303.
@SUPERVISOR JOB VAX01 200 1 8 JOB VAX02 201 1 8 FILMAN VAX02 401 2 8	The job supervisor in machine VAX01. The job supervisor and filemanager in machine VAX02.
@PERIPHERAL LPT 1103 SLOWLPT 1303	Two lineprinters in the current machine (m/c 3) with SPN's %1103 and %1303.
@ROUTE 1503 1101 1101 8	Re-route output for peripheral %1503 to the peripheral %1101.
@DEVICE 3 5002C 101 1400D	Configure logical device 3, as a character input device, supported by line 1 of the multiplexor (for example). Buffer terminates on special characters, echo is on and XON/XOFF is accepted. Line is of even parity, with 7-bit characters and one stop bit, and 9600 baud.
@END	End of parameters.

20.2. OPERATOR MESSAGES AND LOGGING INFORMATION

The system provides for two types of operator messages and logging. Events requiring urgent attention and status information of immediate concern to the operator appear as messages on the operator's console. Thus, for example, requests for a magnetic tape or the allocation/de-allocation of a peripheral would be monitored on the console. More general information about the running and performance of the computer is collected in a log file, which may be examined by the operator. This includes information about the running of individual jobs.

The precise details of the messages and logging information is installation dependent, and on some small machine systems might be of a very limited nature.

20.3. MISCELLANEOUS COMMANDS

1) DRIVE.PERI(I,I)

This procedure may be called by an operator to drive a device on a specific peripheral 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:

```
 __________________
 |    | | | | | | |
 ------------------
      | | | | | |
      | | | | |  --BREAK interrupt from the peripheral
      | | | | |    required after each section.
      | | | | |
      | | | |  ----BREAK interrupt from the peripheral
      | | | |      required after each newpage.
      | | | |
      | | |  ------Transmit XOFF to the peripheral
      | | |        after each newline character.
      | | |
      | |  --------Transmit ETX to the peripheral
      | |          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	%34
Diablo printer (using ETX) (as a document printer)	%2B
Diablo printer (using ETX) (as a lineprinter)	%C
Diablo printer (with XOFF facility) (as a lineprinter)	%0

2) CONNECT(I,I)

This procedure is intended mainly for use on small MUSS systems which might be used as 'terminals' on an alien system. The parameter P1 specifies a peripheral channel and the assumption is that this channel is connected to an interactive port of the alien system. Control is retained by the CONNECT command, subsequent input is routed to the alien system and its responses appear on the user terminal. In addition the CONNECT command implements a primitive control language which facilitates file transfer and downline loading. The command lines begin with the warning character specified by P2 ('*' is the default) and such lines are not transmitted to the alien system. The commands are

*L filename -	this command creates an output stream, with destination = the given filename, and copies all subsequent input received from the alien system to this stream in addition to sending it to the users terminal.
*S -	this command stops the copying of input into the stream created by *L.
*C -	this command causes copying to be resumed.

*T filename –	this command transmits the specified file to the alien system, a line at a time. It waits between lines for a message to be returned which it assumes to be a prompt. All input received during transmission of the file will appear on the terminal.
*E –	this command causes the CONNECT procedure to return control to command level. Any stream set up by the *L command will be broken at this time.

3) 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.

4) 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.

5) 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).

6) LIST.JOBS()

This command allows the operator to list all the processes currently in the system, along with useful information concerning each process. The status of the process is indicated as follows

M – waiting for a message
S – suspended (by suspend command)
T – suspended awaiting termination

7) UPDATE(II,I)

This command allows the operator to

- update a subsection of the system for which only one copy exists on disc (e.g. Fortran Compiler).
- update a subsection of the system which is not the currently selected version, for those subsections which have two copies on disc (e.g. Basic System).

P1	is the name of the subsection and
P2	is the number of the version (where relevant) to be updated.

8) INIT.DISC(I,II,I)

This procedure initialises the disc on the unit number given by P1 according to the mode given in P3. P3 is interpreted as follows:

0 –	temporarily rename the disc with name P2. The disc will revert back to its old name when the pack is removed from the drive.
1 –	permanently rename the disc with name P2.
2 –	initialise the file database on the disc and name it as P2.

If modes 1 or 2 are required, this command should be called prior to mounting the pack on the drive.

21. LIBRARIES

It is clear from the description of MUSS that its operation is very dependent 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 consists of a directory and the associated procedure bodies and possibly data segments. In some implementations these might occupy separate files but the file name of the directory is always regarded as the name of the library. A user need not be conscious of the additional files which constitute a library except that they will appear in his file store, with names derived from the name of the library (e.g. library name 0, library name 1, etc). A complete library whether it be one or more files contains the following

a list of names for the procedures it contains,
their parameter specifications,
a list of the names of the user defined
types used as parameters,
their specifications,
a procedure entry table,
initialisation code,
initialised data structures,
and 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 is always true.

The segment can be recompiled and, providing it contains the same procedures in the same order (in fact it is the order of the procedure names in the export list of the module heading, rather than the order of the procedures themselves) with unchanged specification, previously bound calls will continue to operate. This is still true even if new procedures are added to the end of the export list.

21.2. THE SYSTEM LIBRARY

At the time a MUSS system is generated, a selection of library segments are included as the system library.

The content of this library can be different at different installations, but at least the commands described in this Manual will normally be provided, and extensions beyond this set are usually implemented as private libraries.

21.3. PRIVATE LIBRARIES

The commands LIBRARY and DELETE.LIB were introduced in Chapter 2 of the User Manual. The first dynamically extends the set of procedures available, and the second reduces it, by removing those of the named library.

A library can contain code at the outer module level, and this is treated as initialisation code to be executed at the time of opening a library by means of LIBRARY. Typically this code might create working segments and initialise data structures.

Sometimes it is necessary to discover the content of a compiled library. The following command is intended for this purpose

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

The parameter P1 specifies a library name which must already have been opened (by LIBRARY). P3 indicates the level of detail required as follows

```
P3 = 0 procedure names only
P3 = /= 0 gives procedure names and parameter types
```

P2 may be used to select a subset of the procedures. If P2 is non-zero only procedures whcse names contain the string P2 will be listed.

21.4. PROCEDURES FOR CONSTRUCTING LIBRARIES

The procedures defined in this section are not normally used directly. They are the means by which MUTL creates a library.

1) INIT.DIR(II,I)I

This procedure creates a new empty directory for a library which will have name P1. P2 specifies the maximum number of procedures which are to be added to the library. If P2 = 0 the procedure will assume that a program rather than a library is to be compiled. INITDIR returns the segment number of the directory, and this is used as a parameter of the procedures which add to the library.

2) ADD.PROC(I,[C],[I])I

This procedure allows the specification of a new library procedure to be added to the directory, specified by segment number P1. P2 gives the name of the procedure and P3 is a list of integers which give the MUTL encoding for each parameter. It returns a number which relates to the position of the procedure in the library. After a call of INIT.DIR the first procedure number returned will be 1 and it will thereafter increase by one on each call of ADD.PROC. This number should be used instead of the MUTLN in type specifiers which are pointers to the procedure.

3) ADD.TYPE(I,[C],[I])I

This procedure allows the specification of a user defined type to be added to the directory specified by P1. P2 is a list of integers which give the types of its constituent components in MUTL encoding. The result is a number which relates to the position of the type definition inside the library. Like the procedure number, it will count up from 1 on successive calls, and it must be used instead of MUTLN's in type specifiers.

4) ADD.SEGMENT(I,I,I,I)

This procedure is used to pass into the directory structure information relating to the code and data segments of a compiled library. If the segment exists at the time of the call of ADD.SEGMENT, and it is to be filed, then

P1 specifies the directory segment
P2 gives a compile time segment number
P3 gives a run time segment number
P4 gives the required run time access
(as in OPEN.FILE)

If the segment is to be created as a scratch segment when the library is opened, P1 and P3 should be set as above, but P2 will be zero and P4 will give the required segment size.

The first segment added to a library by this command is assumed to be the principle code segment containing initialisation code and the procedure entry table.

2) CLOSE.DIR(I)

This procedure completes the construction of a library directory and files the directory segment, specified by P1, under the name given in the call to INIT.DIR.

A library directory is typically formed, during compilation, by a call to INIT.DIR, several calls to ADD.PROC, and possibly ADD.TYPE, and finally by calling CLOSE.DIR.

6) MERGE.DIR(II,II,II)

This procedure creates a new directory P3 by combining directories P1 and P2. The code segments of directories P1 and P2 remain separate.

21.5. PROCEDURES FOR IMPLEMENTING CALLS ON LIBRARY PROCEDURES

These procedures are only needed by compilers and command interpreters.

1) FINDN([C],I)I

This procedure searches the lists of names associated with both private and system libraries, for the name specified by P1. P2 indicates whether the name is the name of a procedure (0) or type (1). It returns an integer, called a 'library index'. The value 0 means that the name is not present in the library. Normally the 'library index' is only used as a parameter to other procedures such as FINDP, FINDT and the TL.PL, and users do not need to know the encoding. In fact the encoding is machine dependent..

2) FINDP(I,I,I)I

This procedure is given a library index as P1, an optional procedure number (as returned by ADD.PROC) as P2, and an integer as P3. It returns the type of parameter P3 of procedure P1(P2) in MUTL encoding. If P3 = 0 it returns the number of parameters and if P3 is this number + 1 it returns the result type of the procedure. If P2 is negative the procedure whose number is encoded in P1 is assumed otherwise the procedure number in P2 is substituted.

3) FINDF(I,I,I)I

This procedure is the means by which the MUTL types of the constituent fields of a type are discovered. P1 is a library index as returned by FINDT. P2 is an optional type number (as returned by ADD.TYPE) and P3 specifies a field number. As in FINDP, P2 will be ignored if negative, otherwise it will replace the number in P1. Also if P3 = 0 the number of fields will be returned.

4) LOOK.UP.N(I,[C],[C])I

The purpose of this procedure is to yield information about the procedures and types in the library specified by P2. In particular it yields the library index of the nth name as a result, where n is specified by P1. It also copies the name into P3, preceded by a byte which indicates whether it is a procedure or a type (ms bit = 0/1), and gives the length of the name (ls 7 bits). If the library does not have the specified nth procedure a zero result is returned.

5) LOOK.UP.LIB(I)

This procedure returns information concerning whether segment P1 is a code segment of a currently open library. If this is so the directory name and user name (as obtained from the LIB command) are returned in PWW1 and PWW2 respectively, otherwise PWW1 and PWW2 are both set to zero.

21.6. 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.

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.

23. THE COMPILER TARGET LANGUAGE (MUTL)

23.1. INTRODUCTION

The tasks performed by a compiler in translating a high level language program to some object form may be divided into two parts. Some tasks, such as lexical and syntax processing, are largely independent of the machine on which the compiler is running; while other tasks, such as code generation, are very much more dependent on the type of machine that the program is being compiled for. The concept of a compiler target language is to present an abstract machine model to which the compiler targets its code, thereby increasing the proportion of the compiler which is almost machine independent. Thus a compiler may be considered in the two parts shown below.

```
                                    |
  ---------------------             |     ---------------------
 |                     |            |    |                     |
 | High level language |            |    |  Target language    |
 |  to                 |            |    |  to                 |
 |  target language    |---|-->|    |  machine code       |--->
 |  translator         |Target |    |  translator         | Machine
 |                     | code  |    |                     |  code
  ---------------------             |     ---------------------
                                    |
 <- low level of machine ->|<- high level of machine ->
          dependency        |         dependency
```

The compilers which run under the MUSS system are organised in this way and the target language is the Manchester University Target Language (MUTL).

The justification for this separation of a compiler is mainly threefold.

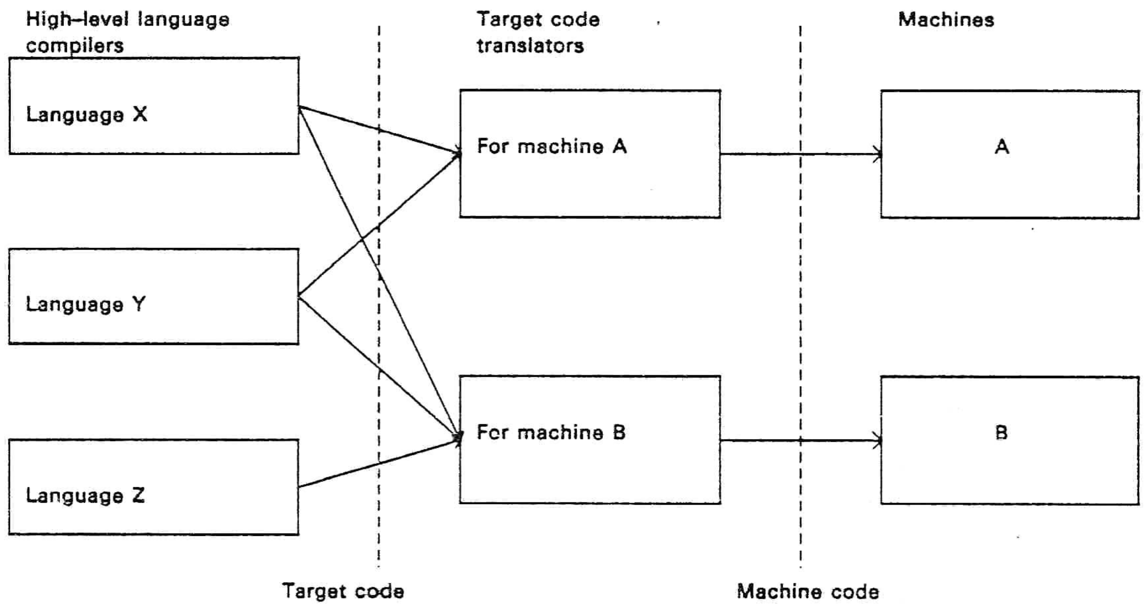
1. The abstract machine model can be the target machine for several high level language compilers, thus all code generation tasks of these compilers is centralised, and they are thereby simplified.

2. A single compiler can produce code for several different machine types, by having several abstract machine code translators available to the compiler.

3. On transferring the compilers to another type of computer, only the abstract machine code translator need be rewritten for the new environment.

Figure 23–1 illustrates points 1 and 2.

Figure 23–1 MUSS Language Structure



Although the main function of a target language is code generation, it may to advantage perform other subsidiary functions. For example, the task of compilation mode control and the provision of high level language run time diagnostics may be functions of the target language.

Experience of the design of several abstract machine models for multi–language multi–machine usage has led to the following conclusions.

1) The design of the target model is strongly influenced by the efficiency of the code required. If the best object code efficiency possible for a multiple register machine is to be achieved the compiler must be aware of the number and type of registers available on a particular machine in order to make optimum use of registers. For single accumulator (2900, MU5) and stack based machines, abstract models based on a single accumulator or stack architecture are equally effective, and such models are capable of reasonably good object code efficiency for multiple register machines.

2) Optimisations such as removing multiple evaluations of sub–expressions and moving invariant expressions outside loops should be in the machine independent part of the compiler and not in the target language translator. As a consequence of this the unit of code generation of the target language can be in terms of single instructions in the abstract model.

3) Compiler target languages which are easily portable normally achieve this by restricting their data types and data structures, consequently these languages often have considerable inefficiency in data storage mapping and inefficient data access code. In

order to obtain efficient data access code on different machines, and allow heterogeneous data structures, the target language must be sympathetic to the type and structure of data.

4) The MUTL abstract model caters primarily for Fortran, Pascal, Algol and Cobol as well as the system implementation language MUSL. An actual implementation of the model may omit some of the capabilities of the model when they are not required for the set of high language compilers required at a particular installation. For instance if only Fortran and Algol are required, the target language need not have decimal arithmetic. As further compilers based on MUTL are developed, then where necessary the MUTL model definition will be revised to encorporate new requirements.

5) The communication between the compiler and the target language should be essentially one way. Restricting the target language in this way means that the high level language to target code translation and target code to machine code translation may, where necessary, be completely separated. There are several advantages of this approach:

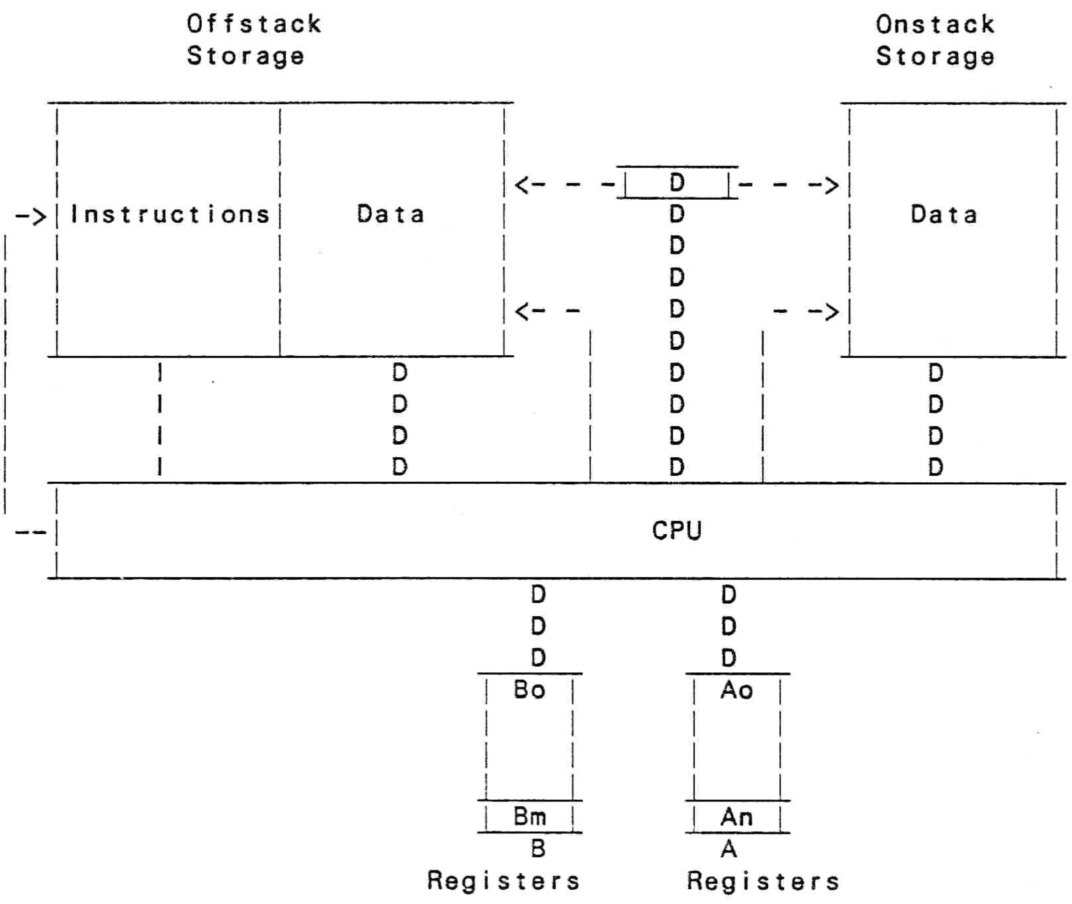
a) an encoded form of the target language (MUBL) can be utilised in self bootstrapping the high level language compilers,

b) in a computer network, compilation for several different machine types may be done on a central machine, and if the output of the compiler is in the encoded target language code, it can be assembled into machine code on the appropriate execute job machine in the network, and

c) for small machines, which have a small address space it provides a natural division of the compiler into two passes.

In the abstract model data is allocated in terms of scalar or vector quantities, and from these basic data components more complex data structures are built. The model provides addressing functions to access such data structures, but the target language provides the declarative functions for the allocation of data in the abstract model. As the target language deals with data at this high level, data accessing on the actual machine can be handled efficiently.

The diagram below illustrates a simplified view of the MUTL model, it basically consists of registers, stores and a CPU.



DDDDDDD data paths
- - - - store requests
IIIIIII instruction path

23.1.1. Registers

As mentioned briefly in the introduction, optimum register usage on a multiple register machine is best achieved if the compiler can adapt to the number of different registers available. Thus the MUTL model has multiple B registers and multiple A registers, but only a single D register. At present all issued versions of MUTL implementations only cater for one B register and one A register.

The A registers operate in one of several modes:

integer
unsigned integer
floating point
fixed point decimal
pointer
generic

The B registers are primarily for subscription of data structures. They operate in integer mode.

The D is a data selection register that references into a data structure in order to access a component from it.

23.1.2. Arithmetic precision.

To allow MUTL to be implemented on micro, mini and mainframe computers the precision of an A register is selected from the following:

1, 8 and 16 bits (with integer mode only)
32, 64 and 128 bits (all modes of A except decimal)
any byte sized multiple up to 80 bits (decimal).

An implementation of MUTL need only implement sufficient arithmetic capability for the high level languages concerned. B register arithmetic is of integer mode of sufficient precision for data subscription purposes.

23.1.3. Instruction and Data Stores.

The model incorporates off-stack storage and on-stack storage. Instructions are fetched in sequence from the off-stack store and executed in the CPU, until a control transfer order is executed at which point instruction fetches continue from some other 'labelled' place in the store.

Data is contained in both stores. The off-stack store generally contains data that does not belong to any particular procedure e.g. Fortran Common, and permanent data of a procedure e.g. OWN in Algol, SAVE in Fortran. The on-stack store holds the data of all the currently active procedures. Within the stack frame of a procedure, its data is split into three groups:

a) parameters of the procedure
b) variables (scalars, vectors and aggregates)
local to the procedure
c) partial results stored in a LIFO basis.

23.1.4. Operand types and lengths.

Data in store may be considered as a sequence of simple typed operands, where each operand is either arithmetic or a pointer.

An arithmetic operand is either:

a) 1 bit long
b) or between 1 and 16 bytes long
c) a string of up to 32 bits.

At operand declaration time type is specified. Arithmetic operands may be of real, signed or unsigned integer and decimal type. A 1 bit operand is always of unsigned integer type, real operands are restricted to precisions of 2, 4, 8 and 16 bytes, and decimal operands is limited to 10 bytes. In addition bit string operands are restricted to signed and unsigned integer types.

Pointer operands refer to: data items, labels and procedures. However, the information content required for such entities differs between high level languages, consequently six types of pointers are distinguished.

a) pointer to a data item
b) bounded pointer to a data item
c) pointer to a label
d) pointer plus environment information of a label
e) pointer to a procedure
f) pointer plus environment information of a procedure

23.1.5. The CPU.

The instruction set is mainly one address and most instructions consist of a function part and an operand part. It is fully described in Section 23.9. Implementors of MUTL code generates should beware that whilst many MUTL instructions translate into one machine instruction, and some may translate into several machine instructions, others such as those concerned with operand selection and arithmetic mode selection are not expected to generate any code.

23.2. OVERVIEW OF MUTL CONCEPTS

In addition to providing the imperative functions of code generation for compilers, MUTL provides declarative functions for data items, procedures, labels etc. Before a specification of the procedural interface is given, it is necessary to introduce briefly; compilation units, storage control, the MUTL concept of types, names, machine code efficiency, diagnostic information and MUTL code output.

23.2.1. Compilation Units (Modules)

The basic unit of compilation is a module which normally contains one or more procedures, and a program or library may involve several such modules. A module contains the specifications for the entities which are considered local to the module. Some of these may be indicated as 'exports' in which case they are available for import into other modules. In order that modules may be compiled independently they must contain specifications of their imports, which at compile time are taken on trust. However, these import specifications are passed to MUTL, so that at load time it can check their consistency with the corresponding export specifications. The entities referenced in import or export specifications are called interface entities. Only areas of store, data items, data types, literals, procedures and subroutines, and labels may be interface entities. In fact areas of store are slightly different from other interface entities in that they do not belong to any particular module, but are shared between all modules using them.

It is the symbolic name of the interface entity, specified in characters, that enables inter-module references to be resolved. Therefore, each interface entity of one kind must be uniquely identifiable by its symbolic name.

23.2.2. Storage control

As discussed earlier storage is split into off-stack and on-stack storage. On activation of a dynamic procedure, a new stack frame is created to contain its parameters and local variables. Further space on the stack is obtained as required at run time by the TL.MAKE function. The off-stack store is considered by MUTL as a number of areas into which code is planted and data is statically allocated. There are 255 such areas numbered from 1 upwards, from which current code and data areas are selected as appropriate. The amount of store allocated in an area while selected as a current area is known as a partition. The selection of an area as current defines the start of a partition, and the selection of another area terminates the partition. An area thus consists of one or more partitions.

The procedure TL.S.DECL is used to statically allocate off-stack and on-stack data. A portion of an area (off-stack or on-stack) may be statically allocated as a SPACE so that at run time variables may be dynamically allocated within it, this is achieved by the TL.MAKE function.

The organisation of the stack is the MUTL translators responsibility. However, non-local accesses will generally involve static-links, or a display approach, or even a mixture of both methods.

23.2.3. Data types

In MUTL there are a set of predefined data types known as the basic types. Additional types known as aggregate types may be specified. These consist of a concatenation of several basic types or previously defined aggregate types. A variable of aggregate type may, when required, be considered as a single entity. In general aggregate types may be considered as a hierarchic structure of types. For variables of aggregate types, individual fields are identified by their relative position, counting from zero (at each level in the hierarchy). Within an aggregate type, fields may have several alternative type definitions, thus allowing variables to contain variant fields.

Basic types are classified into arithmetic and pointer types. An arithmetic basic type specifies size and mode information (e.g. 16 bit integer) for the data item. Pointer types specify whether the type of the data item referenced is a data, label or procedure item. There is also a typeless data pointer, but whenever a pointer of this kind is loaded into the D register, type information must first be given by calling TL.D.TYPE.

23.2.4. Allocation of names

MUTL names are used to identify and reference many different kinds of entities, for example data variables, literals, procedures, type specifications, etc.. In order to maintain the one way communication of MUTL, for reasons specified earlier, MUTL and the compilers use the following simple algorithm for allocating names. Names exist as numbers in the range 2 to 1983. Names are allocated consecutively from 2 as entities are declared. At the end of procedures and blocks all names allocated within the procedure or block become undefined, and they are re-assigned consecutively as new declarations occur.

Subject to the non-local accessing restrictions of variables and labels of procedures (discussed later in section 23.7) all names currently defined can be referenced regardless of the level of procedure nesting at which they are defined.

23.2.5. Code efficiency and optimisation

The optimisation task is shared between compilers and the MUTL translator. It is the compilers job to perform global optimisation techniques, and to produce optimum MUTL code sequences. The MUTL translator then applies peephole type optimisation when generating code from those sequences.

As mentioned earlier, the strategy in utilising multiple registers is to have optimising compilers tune their output code to the number and type of register available. Thus the procedure TL.ENQ.REG(23.12) supplies this kind of information about the compiling environment. Normally a MUTL translator reserves some actual machine registers for such tasks as stack management, operand conversion and data accessing, the remaining machine registers being nominated as MUTL registers. Furthermore to assist efficient code generation the use of MUTL registers is deterministic, this is achieved with assistance from the compiler.

On some actual machines the intelligent use of base registers reduces significantly the size of operand access code (for example, on the VAX computer when a base register is loaded to a static area then access to an operand relative to the base register costs 2 bytes, while an absolute operand access cost 5 bytes). However, when the number of base registers is limited then global analysis is necessary in determining an optimal strategy for base registers. Thus there is a diochotomy in that it is the compiler's role to perform global analysis but as the operands in the MUTL interface are at a high level, code generation details of operand accesses are a MUTL function. The solution to this is to have a set of MUTL base registers, and as with A and B registers the number of base registers is dependent on the actual machine and is obtained by calling TL.ENQ.REG. A MUTL base register is loaded either with the address of a MUTL segment or the address of a frame on the stack, this is done under the guidance of the compiler.

To use base registers effectively for accessing non-local operands on the stack the compiler needs to cater for static-link and display based stack organisations. A compiler should not load a base register for accesses within the currently active frame or any frames covered by the display; the procedure TL.ENQ informs the compilers of how many procedural textual levels are covered by the display. In accessing an operand covered by a base register MUTL may then plant an access relative to a base register.

23.2.6. Program error detection and diagnostics.

The checking for possible program errors is normally done by a mixture of hardware and software. MUTL normally provides some of the software checking, and control of this checking by compilers may be required. The MUTL procedure TL.CHECK provides this control.

The run time diagnostic procedures in MUTL provide primitives to support machine independent diagnostic software for such tasks as producing compile and data maps, debugging (batch and interactive), and profiling. Therefore source language reference information such as data item symbolic names, procedure symbolic names and source line numbers are passed into MUTL during compilation, this and the relevant declarative information is retained at the end of compilation if run time diagnostic support is required.

23.2.7. MUTL Code Output.

The forms of code output produced by MUTL depends largely on the requirements of a particular installation.

A compilation produces either all or part of a program or a library. When compiling a library, a directory is automatically generated for the libraries interface. Apart from the directory structure, programs and libraries are similar. The code at the outer textual level of a compiled library (i.e. not embedded in a procedure) is executed, as initialisation code for the library, whenever it is loaded. The definition of library and program files in achieved by MUTL calling procedures in the Library Organisation section of the MUSS Library.

MUTL views a program or library as a collection of MUTL segments, these segments are numbered 0, 1, 2 etc. The model of a program is of MUTL segment 0 containing code while other segments normally contain data, but may contain code. Entry to a program is always via MUTL segment 0, therefore any entry and exit sequence code concerned with running a program is added to segment 0 by MUTL. The model of a library is of MUTL segment zero containing a library entry table, followed by code for the library procedures. There is a directory structure created for the library, but its placement and organisation is dependent on the library organisation for a particular machine. Entry to the library initialisation code is always via MUTL segment 0, therefore, any entry and exit sequence code necessary is added to segment 0 by MUTL.

Many situations arise where it is necessary for the user to exercise control over the mapping of the module areas into actual storage. Examples of this need are in overlaying for machines with limited address space, and the creation of operating systems where strict control of the placement of areas is required. In some high level languages such control is a language feature, while in others, where it is necessary, control is provided either by compiler directives or by job commands. The procedures in 23.4 enable the compiler to control the placement of compilation output.

23.3. TYPE DEFINITIONS

The procedures in this section enable aggregate types to be defined and a MUTL name allocated. In other contexts where the type of an entity has to be given, a MUTL TYPE SPECIFIER is used. This is an encoded 14 bit integer whose least significant two bits indicate whether the specifier relates to

0	an entity of the given type
1	a pointer to an entity of the given type
3	a bounded pointer to a vector of entities of given type.

The other part of TYPE SPECIFIER either defines a basic type or the MUTL name of a user defined type. Basic types have values which are either less than 64 or greater than 2047 (shifted left two places to accommodate the two bits defined above). Thus in order that they may be distinguished user types are actually represented by their MUTL name + 64 (again shifted left two places). Thus a TYPE.SPECIFIER for basic types whose size is specified as one or more bytes is encoded as:

Bits 2–5	specify the size of the type as the number of bytes less one (e.g, 1 means a 2 byte type)
Bits 6–7	specify the type mode. 0 – real 1 – signed integer 2 – unsigned integer 3 – signed fixed point decimal with a representation of a decimal digit per nibble with one nibble for the sign, thus a 7 byte decimal represents a signed 13 decimal digit number.
Bits 8–13	0

As real types are restricted to 2, 4, 8 and 16 bytes, the above leaves 48 spare encodings. Some of these are utilised as follows:

```
%20 - denotes a 1 bit scalar

%01 - Typeless data pointer.
%03 - Typeless unbounded data pointer.

%24 - Pointer to a procedure.
%28 - Pointer to a procedure and its environment.
%2C - Pointer to a label.
%30 - Pointer to a label and its environment.
```

A TYPE.SPECIFIER for basic types whose size is specified as a bit string is encoded as

Bits 2-5	0
Bits 6-7	specify the type mode 1 - signed integer 2 - unsigned integer
Bits 8-12	specify the size of the type as the number of bits less one (e.g. 2 means a 3 bit type).
Bit 13	1

1) TL.TYPE([SYMB.NAME],NATURE)

This starts the specification of an aggregate type and allocates a MUTL name for the type, unless one had previously been allocated for it. Type specifications themselves may not be nested, but an aggregate type may include previously defined types and pointers to types which are not yet fully specified. The constituent types of the aggregate are added by calling TL.TYPE.COMP repeatedly. During the specification of a type only calls to the MUTL procedures TL.TYPE.COMP and TL.END.TYPE are permitted.

The second parameter specifies whether the type is internal to the module, an export from the module, or an assumed type from another module. A type may be imported without its type detail being re-specified in this module in which case there are no further calls to TL.TYPE.COMP or TL.END.TYPE; this situation arises when the language compiler need not be aware of the type details in compiling a module.

An incremented type is an aggregate type where the type specification is given incrementally over several modules. For an incremental type a skeleton specification is first given for the type so that MUTL knows it's overall size for code generation purposes. The actual specification is built up incrementally by modules adding fields to the type. If a module adds fields to an incremental type it cannot access directly any of the other fields added by other modules.

Note that a type specification containing no fields is permitted.

The position of a component, counting from zero, within the type definition is used in accessing it.

Parameters:-

[SYMB.NAME]	Bounded pointer to a byte vector containing the symbolic name in characters of the type. When the type is an interface entity the symbolic name identifies the type, it is also retained in the symbol table for run time diagnostic purposes.

NATURE	Bit 14 = 1 type exported from module. Bit 15 = 1 type of an assumed entity to be imported.
Bits 0–13	%0000 means a MUTL name is to be allocated and its type specification follows immediately. %0001 is as %0000 except that the type has no alternatives and contains only one field. In referencing this type selection of the contained field is automatic, i.e. no SEL FLD is necessary, such a type is known as an automatic type. This feature enables compilers to create extra types to assist in type checking and for defining enumeration types. %1000 means a MUTL name is to be allocated and its type specification is not specified in this module; MUTL obtains the specification from the export of the type from another module. %2000 means a MUTL name is to be allocated but its type specification is given later in the module. %3000 indicates fields are to be added to an incremental type. A MUTL name is allocated and further fields are added to the type's specification by calling TL.TYPE.COMP to add each new field, and then finally calling TL.END.TYPE. Within the module the newly added fields are referenced by ascending field numbers 0, 1, 2 etc, regardless of how many fields there are already in the type. %0002 to %07FF means the type specification now follows for the MUTL name specified in bits 0–10, which had previously been allocated with %2000 specified for this field. %3002 to %37FF is as %0002 to %7FF except that fields are to be added to an incremental type as in %3000 above.

Example:

To create a type T consisting of a vector of two integers (4 bytes each) TVI, and a vector of three reals (8 bytes each) TVR.

```
TL.TYPE( ["T"],0)
TL.TYPE.COMP(%4C,2,["TVI"])
TL.TYPE.COMP(%1C,3,["TVR"])
TL.END.TYPE(0)
```

A type S consisting of an integer (4 bytes) SI, and a ten element vector SVT where each element is of type T can then be created by:

```
TL.TYPE( ["S"],0)
TL.TYPE.COMP(%4C,0,["SI"])
TL.TYPE.COMP(MUTL NAME of T*4+256,10,["SVT"])
TL.END.TYPE(0)
```

The ten element vector is referenced as the second component of S (i.e. SEL FLD 1).

2) TL.TYPE.COMP(TYPE,DIMENSION,[SYMB.NAME])

Adds the next component to the current aggregate type specification. The first two parameters specify the type of the component.

Parameters:-

TYPE -	Bits 0-13 give a TYPE SPECIFIER as described above at the start of 23.3. Bit 14 a value of one means that P3 specifies a list of names for an enumeration type.
DIMENSION -	0 scalar >0 vector, parameter specifies number of elements. <-1 vector, the parameter is the negated MUTL name of a literal whose value gives the vectors dimension.
[SYMB.NAME] -	Bounded pointer to a byte vector containing the symbolic name in characters of the component, or a list of symbolic names for the names of an enumeration type (each name in the list is terminated by a zero valued byte). This parameter is for run time diagnostic purposes only.

3) TL.END.TYPE(STATUS)

Terminates the addition of fields to the aggregate type currently being defined. STATUS indicates the end of a type specification, the end of an alternative within the type, or that the type specification is the skeleton of an incremental type.

Parameters:-

```
STATUS 0 - end of type specification, or end of fields
           being added by this module to an incremental type.
STATUS 1 - another alternative type specifications follows
       2 - indicates a skeleton specification for an
           incremental type.
```

23.4. STORAGE MAPPING

As previously explained in 23.2 MUTL plants code and literals within the current code area and allocates statically mapped storage within the current data area. All areas are placed in MUTL segments. The procedure TL.SEG introduces a MUTL segment into the compilation, while the procedure TL.LOAD specifies into which MUTL segment a particular area is placed. Thus a modules areas may be mapped to several MUTL segments and a MUTL segment may contain areas from several modules. The characteristics of a MUTL segment are that it occupies, when in store at run time, a contiguous address space and all parts of it have the same level of protection (e.g. same level of protection and overlay type) and similar items throughout the space are accessed in an identical manner. Note that if there is more than one area mapped to a MUTL segment the partitions of an area may not be allocated contiguously within the MUTL segment, and generally the currently selected data and code areas should not be mapped to the same segment as interleaving may occur of code and data. Normally each data declaration can be considered as unrelated from other data declarations, but for off-stack data involved in a Fortran type EQUIVALENCE it implies a specific ordering of the data. Therefore the TL.EQUIV.POS is provided to permit the declaration point of a variable to be specified explicitly within an off-stack data area.

1) TL.SEG(SEG.NUMBER,SIZE,RUN.TIME.ADDR,COMP.TIME.ADDR,KIND)

Introduces a segment into the compilation. These are known as MUTL segments and have numbers in the range 0 to 31. On introducing MUTL segment 0 initialisation code concerned with compiling a program or library is planted automatically by MUTL.

Parameters:–

SEG.NUMBER	Number of MUTL segment.
SIZE	Size of MUTL segment in bytes. On machines where control transfers between virtual store segments is severely restricted (e.g. procedure calls only) the maximum size of a MUTL segment is normally limited to that of a virtual store segment, whereas for other machines a MUTL segment may span several consecutive virtual store segments. A size of zero means a segment of default size is created.
RUN.TIME.ADDR	of MUTL segment specified in bytes. A value of –1 means that the segment's address will be allocated automatically.
COMP.TIME.ADDR	of MUTL segment specified in bytes. A non negative value specifies the compile time address for the MUTL segment. A value of –1 means the compile time address will be automatically allocated; a value of –2 means that the MUTL segment need not be allocated at compile time as no code production or static data initialisation occur for this MUTL segment; and a value of –3 is similar to –2 except that on loading a program or library for execution the MUTL segment is not introduced automatically by the loading mechanism into the execution environment.
KIND	This parameter specifies the properties of the MUTL segment. Bit 1=1 means that the MUTL segment at run time requires execute access. Bit 2=1 as Bit 1 but for read access. Bit 3=1 as Bit 1 but for write access.

2) TL.LOAD(SEG.NUMBER,AREA.NUMBER).

This procedure specifies into which segment the partitions of an area will be mapped. The mapping of an area must be given before the area is used. If a partition of an area is nominated as an interface entity, then the mapping may be overridden by a previously specified mapping in another module.

When modules are bound together then once an area segment mapping is established it applies to all modules following until a new mapping is given.

Parameters:–

SEG.NO	Segment number.
AREA.NO	Area number of current module, if between modules then of next module ccmpiled.

3) TL.CODE.AREA(AREA.NUMBER)

Nominates the current code area and sets the current position to the next available location within it.

Parameters:–
AREA.NUMBER

4) TL.DATA.AREA(AREA.NUMBER)

This procedure nominates the current data area and sets the current position within the area. Statically allocated variables are mapped to the current position in the currently selected data area.

Parameters:–

AREA.NUMBER	– 0 stack frame of the current procedure – Area number

5) TL.EQUIV.POS(POSITION)

This procedure resets the current position of the current area to the position specified. The data area to which it applies must be off-stack.

Parameter:–

POSITION:	Position from start of area specified in bytes.

6) TL.INT.AREA(AREA.NUMBER,[SYMB.NAME])

This procedure nominates the area as an interface entity. It is called prior to the area being selected as the current data area.

Parameters:–

AREA.NUMBER	Area number
[SYMB.NAME]	Bounded pointer to a byte vector containing the symbolic name in characters of the area.

23.5. STATIC DATA STORAGE DECLARATION

Variables declared at the outer level of a module are available to all procedures within the module. For variables declared within a procedure, the kind of procedure determines whether its variables may be accessed non-locally.

1) TL.SPACE(SIZE)

Space is statically allocated in the current data area, which could be either on-stack or off-stack, and an integer variable is declared to keep track of the run time usage of the allocated space. A MUTL name is allocated for the SPACE and this name can be used in imperative functions (e.g. TL.PL) to control usage of the space. Structures are created dynamically in the space by the TL.MAKE procedure.

Parameters:–

SIZE	When positive it specifies the size in bytes of the SPACE When negative it is the negated MUTL name of a literal whose value gives the size of the SPACE in bytes.

2) TL.S.DECL([SYMB.NAME],TYPE,DIMENSION)

This procedure declares a variable which may be a scalar or a vector of basic or aggregate type. A MUTL name is allocated for the variable. For variables of a known dimension then, unless the declaration is for an assumed interface entity of another module, storage is allocated for it within the currently selected data area. For variables of an unknown dimension then it is the set of values that statically initialise the variable that determines its size.

Parameters:–

[SYMB.NAME] –	Bounded pointer to a byte vector containing the variables symbolic name in characters. It is used to identify the interface entity and for run time diagnostic identification.

TYPE -	Bits 0 - 13 is a TYPE SPECIFIER, see 23.3 for its encoding. Bit 14 = 1 Data item to be exported as an interface entity Bit 15 = 1 Declaration of an assumed interface entity imported from another module
DIMENSION	0 Scalar > 0 Vector dimension defined explicitly -1 Dimension of initialised vector unknown. Value initialisation determines actual dimension. Storage is allocated at initialisation time. -4096 Initialised scalar with storage allocated at initialisation time. < -1 vector, parameter is the negated MUTL name of a literal whose value gives the vectors dimension.

3) TL.V.DECL([SYMB.NAME],POSN,PRE.PROC,POST.PROC,TYPE,DIMENSION).

This procedure enables MUSL V-store variables to be defined and MUTL names allocated to them. After declaration they may then be treated by MUSL as ordinary variables in imperative statements. A V-store variable is a special control/status register of the MUSS ideal machine (refn. MUSS VOL 1) that can in general be read or written. If the real machine has an actual control/status that is exactly equivalent, the V-store variable may be mapped directly to the actual register. If the correspondence is not exact then the V-store is mapped onto an ordinary store line. In this case it may be necessary to update the store line before a read access and to propagate the implied actions after a write access. Therefore, associated with each V-store of the ideal machine there is, a store line address, and optionally a procedure (PRE.PROC) to be called before a read access, and optionally a procedure (POST.PROC) to be called after a write access.

The read and write subroutines associated with V-store variables which are vectors generally need to know the index number of the element to be acted upon. Therefore, on entry to such procedures MUTL makes the index number available. Within the subroutine the index number is obtained by specifying an operand of %1002, known as V.SUB, in the operand description of TL.PL.

In the implementation of index number passing, whenever the actual machine architecture permits it, the index should be passed in the register that corresponds to the B register in the MUTL model, and an implementor of MUTL may assume that any B register use within a PRE.PROC or POST.PROC destroys the passed index value.

Like other data variables V-store variables may be interface entities.

Parameters:-

[SYMB.NAME] -	Bounded pointer to a byte vector containing the variables symbolic name in characters. It is used solely to identify the interface entity.
POSN	Actual store line associated with V-store variable. This may either be a previously declared variable or the address of a store line. Thus the parameter may either be the MUTL name of a variable or a literal, or a zero which means the current literal gives the store line address.
PRE.PROC	MUTL name of the subroutine to be called before a read access. A value of zero means no subroutine call on a read access.

POST.PROC	MUTL name of the subroutine to be called after a write access. A value of zero means no subroutine call on a write access.
TYPE	Bits 0–13 is a TYPE SPECIFIER, see 23.3 for details. Bit 14=1 V–store variable to be exported as an interface entity. Bit 15=1 V–store variable to be imported as an interface entity.
DIMENSION	= 0 V–store is a scalar variable. > 0 V–store is a vector variable, the parameter specifies the dimension. –1 V–store is a vector variable but its dimension is unknown, this is only permitted on imports. < –1 V–store is a vector variable, the parameter is the negated MUTL name of a literal whose value is the vector's dimension.

4) TL.MAKE(SPACE,TYPE,DIMENSION)

This procedure plants code to create at run time a mapping for a variable either at the top of the current stack frame or in a previously declared space. For the latter the integer control variable of the SPACE is updated to the next available storage location. Yielded in the D register is a pointer to the variable.

Parameters:–

SPACE	The MUTL name of a SPACE (i.e. SPACE > 1) or zero means allocate at the top of the stack frame.
TYPE	TYPE SPECIFIER, see 23.3 for its encoding.
DIMENSION	A value of zero means make a scalar variable and yield in the A register an unbounded pointer to it. A non zero value means make a vector variable and yield in the A register a bounded pointer to it. A positive DIMENSION specifies the number of elements that the vector is to contain, whereas a value of –1 means that the value of the B register on 'making' the vector determines the number of elements in the vector.

5) TL.SELECT.VAR( ).

This procedure notionally declares a SELECT variable and allocates a name for it. A SELECT variable is used to save data structure selection information held in the D register for later use. This variable may only be used in D= and D=> instructions. In a selection instruction sequence involving SEL EL, SEL FLD and SEL ALT, a 'D=> SELECT variable' saves the current selection information of D. A 'D= SELECT variable' reloads into the D register the previously saved selection information.

Before a 'D= SELECT variable' instruction is planted in the code a corresponding D=> to the same SELECT variable must have already been planted. During run time of the code on executing a 'D= SELECT variable' instruction, the SELECT variable must have previously been assigned by the immediately preceding D=> to the same SELECT variable in the code; in other words, information in a SELECT variable has a static scope.

6) TL.SELECT.FIELD(BASE,ALTERNATIVE,FIELD)

This procedure allocates a MUTL name for a field within a data structure pointed to by the BASE parameter.

Parameters:

BASE	MUTL name of a Select variable which specifies the containing data structure of the required field.
ALTERNATIVE	If the containing data structure is of union type, this specifies the alternative, counting from zero, required.
FIELD	Number, counting from zero, of field required.

7) TL.SET.TYPE(MUTL.NAME,TYPE)

A variable or parameter may be declared to be a pointer of typeless type. This procedure is used to set the type of the variable or parameter.

Parameters:–

NAME	MUTL name of variable or parameter.	
TYPE	Bits 0, 1	1 Scalar instance of a type 3 Vector instance of a type
	Bits 2–13	encoded as in 23.3.

8) TL.ASS(MUTL.NAME.OR.TYPE,AREA.NUMBER).

This procedure is used to nominate a variable or a user defined type for the current user defined type literal to which value assignment procedures refer. During value assignments there is the concept of a current assignment position. Normally value assignments start at the first component of the item, but any point can be selected as the current assignment position by calling TL.ASS.ADV appropriately.

If the variable was declared in TL.S.DECL as an vector of unknown dimension, then storage is allocated for the vector when initialisation commences, and the size of the vector is then determined by the set of values assigned. For such variables MUTL assumes that complete value assignment occurs at the same time.

Parameter:–

MUTL.NAME.OR.TYPE	If bit 14 is zero then bits 0–11 is a MUTL name of an off-stack data variable. If bit 14 is one then bits 0–13 is a TYPE SPECIFIER, see 23.3 for its encoding, of the current user defined type literal.
AREA.NUMBER	This specifies the area number in which to allocate storage for vectors of unknown size. A value of –1 means allocate in current code area, and –2 means allocate in current data area.

9) TL.ASS.VALUE(NAME,REPEAT.COUNT)

Assigns the values as specified by the NAME parameter to the components starting at the current assignment position within the literal or variable being assigned to, and advances the current assignment position accordingly. The second parameter allows this to be repeated.

Normally the NAME represents a single value, the exception to this is when the components being assigned to have a size of one byte, in which case the NAME can specify the current basic type literal which had previously been defined to represent multiple byte size values (see Section 23.6).

Parameters:–

NAME	The type of name permitted is dictated by the type of the current component having its value assigned. For components of arithmetic basic type, the name is that of a previously defined literal, and a NAME of zero means the current basic type literal. It is the type of the literal that dictates how the component is assigned a value. This means that logical literals are right justified within the component, with zero padding and truncation where necessary; similarly for integer literals, but with sign extension instead of zero padding; and for real literals, generally, these are left justified with zero padding and truncation where necessary. For components of label type, the name is that of a label, or the current literal with a null label pointer value. For components of procedure type, the name is that of an actual procedure or the current literal with a null procedure pointer value. For components of pointer type, the name is that of an statically allocated data item, in which case the assigned value is a pointer to the data item, or the current literal with a null data pointer.
REPEAT.COUNT	Number of times the values associated with the first parameter are assigned. If this parameter is positive the parameter itself is the repeat count, else a negative parameter indicates that the repeat count is that of a named literal in which case the parameter is the negated MUTL name.

10) TL.ASS.END().

This procedure is called to indicate the end of assignments.

11) TL.ASS.ADV(NO).

This procedure advances the current assignment position over the specified number of basic type components of the literal or variable being initialised.

Parameter:–

NO	Number of basic type components to be advanced over.

23.6. LITERAL DECLARATIONS

The procedures in this section provide for the declaration of literals. First a value must be assigned to the current literal, then a MUTL name is assigned to the current literal. This two stage approach is necessary because the current literal is used in other contexts. Literals of basic type and user defined type are defined in this way. There are two current literals, one of basic type and one of user defined type.

1) TL.C.LIT.16(BASIC.TYPE,VALUE.16)

2) TL.C.LIT.32(BASIC.TYPE,VALUE.32)

3) TL.C.LIT.64(BASIC.TYPE,VALUE.64)

4) TL.C.LIT.128(BASIC.TYPE,VALUE.128)

These procedures are used to define the value of the current basic type literal. The first parameter specifies the type of the current literal. The size of the basic type value can be smaller than the size of the value parameter, in which case the value is right justified within the parameter.

Parameters:–

BASIC TYPE –	Bits 2–7 specify a Basic arithmetic scalar type for the value. These are encoded as bits 2–7 of basic type specifications. In a cross compiling situation conversion of character values will be necessary when the compiling and target machines have a different character set. Compilers indicate character values by setting Bit 0. Bit 0 = 1 means value specified as a character string. Bit 0 = 0 means value in binary.
VALUE.16 – VALUE.32 VALUE.64 VALUE.128	Value of current literal

5) TL.C.LIT.S(BASIC.TYPE,[VALUE]).

This procedure provides an alternative way of defining the current basic type literal value. To expedite the initialisation of byte vectors, this procedure also permits the 'current literal' to be a sequence of byte sized values.

Parameters:–

BASIC.TYPE	See BASIC.TYPE of above procedures TL.C.LIT.16, etc...
[VALUE]	A bounded pointer to a byte vector. If the literal type does not have a size of one byte the current literal consists of a single VALUE, the size in bytes of the literal type and VALUE must be the same. Otherwise the length in bytes of the [VALUE] parameter determines the number of byte sized values associated with the current literal.

6) TL.C.NULL(TYPE)

This procedure assigns a 'null' pointer value to the current basic type literal.

Parameters

TYPE	Specification of a pointer type, see 23.3 for its encoding.

7) TL.C.TYPE(BASIC.TYPE,KIND.TYPE.ALTERNATIVE).

This procedure defines the current literal whose type is specified by the first parameter with information about a data type. Zero in KIND sets the current literal to the size in bytes and a one in KIND sets it to the boundary alignment in bytes of a data type, whose MUTL name is specified by TYPE. The fourth parameter yields information about an alternative by setting the parameter to the required alternative (counting from zero). A negative value in ALTERNATIVE yields information about the complete type.

Parameters:–

BASIC.TYPE	See 23.3 for its encoding.
KIND	Specifies information required.

TYPE	MUTL name of type.
ALTERNATIVE	Specifies number of alternative.

8) TL.LIT([SYMB.NAME],KIND).

This procedure allocates a MUTL name for a literal which has the value and type of the current literal. For literals of basic type then the above procedures 1) to 6) specify the literal value. For literals of user defined type then the TL.ASS procedures 9) to 11) specify the literal value.

For imported literals the value is optional. When the value is present it is checked against the exported literal value.

Parameters:-

[SYMB.NAME]	Bounded pointer to a byte vector containing the literal symbolic name in characters. This is also used if the literal is an interface entity.
KIND	Bit 14=1 Literal to be exported as an interface entity. Bit 15=1 Literal to be imported as an interface entity. Bits 0-13 A value of zero means it is a basic type literal and a value of 2 means it is a user defined type literal, in both these cases the literal value is that of the appropriate current literal. For imported literals any other value is interpreted as a TYPE SPECIFIER, see 23.3 for its encoding, of the literal and no value is given.

23.7. PROGRAM STRUCTURE DECLARATIONS

There are basically three types of program structures; namely procedures, subroutines and blocks.

A procedure may have parameters and a result. Procedures are further classified with regards to the permitted non-local availability of their variables, and also whether a procedure is static or potentially recursive. The local namespace is created dynamically on entry to a potentially recursive procedure. The non-local accessibility of a procedures variables is at one of the following levels.

I) No non-local variable access.

II) Non-Local variable access permitted.

If the procedure kind is not specified explicitly, then by default its variables may not be referenced non-locally.

A subroutine has no parameters and no stack frame of its own, but it may have a result and be called recursively. For subroutines that are only invoked from the code of the immediately enclosing procedure the code of the subroutine may access (as local variables) the variables of the enclosing procedure, for all other subroutines such access is prohibited.

The same set of procedures are used to define procedures and subroutines.

A block is a delimited sequence of code. The block has no name and is not referenced from control instructions. Any labels or variables declared within the block are only accessible within the block, but the variables of the enclosing procedure, if there is one, are still accessible as locals from within the block.

1) TL.PROC.SPEC([SYMB.NAME],NATURE)

A specification must be given before a procedure (or subroutine) is either defined or referenced. A procedure (or subroutine) specification consists of one call to TL.PROC.SPEC, followed immediately by the appropriate number of calls to TL.PROC.PARAM to specify its parameters, and terminates with a call to TL.PROC.RESULT which indicates the end of the parameters and also specifies the result type. Of course for a subroutine specification there are no calls to TL.PROC.PARAM.

A procedure (or subroutine) definition starts with a call to TL.PROC. For procedures its kind is specified, unless the default kind of procedure is required, by calling TL.PROC.KIND before the first imperative instruction of the procedure is planted. Finally, TL.PROC.END is called at the end of the procedure (or subroutine).

The symbolic names of parameters, these are used for diagnostic purposes only, are supplied to MUTL by calling TL.PARAM.NAME during the procedure definition, i.e. in between calls of TL.PROC and TL.END.PROC.

This procedure normally allocates a name for the declared procedure (or subroutine).

Parameters:-

[SYMB.NAME]	Bounded pointer to a byte vector containing the characters of the procedures (or subroutines) symbolic name. It is used to identify the procedure (or subroutine) if it is an interface entity, for runtime diagnostic identification, and for constructing a directory of a library if this procedure is part of its interface. To permit abbreviated names to be used to identify library procedures, then the name supplied to MUTL contains both lower and upper case characters. The abbreviated name consists of these upper case characters. The only use of this abbreviated name is in constructing the directory of the library, and for interface and diagnostic purposes the name is handled as if it were all upper case characters.
NATURE	Bit 0 A value of zero means the procedure is dynamic i.e. a potentially recursive, and a one means it is static. Bit 1 A value of zero means this is a procedure specification, and a one means it is a subroutine specification. Bit 3 A value of one means this is a library procedure specification. Bit 13 A value of one means that a MUTL name is required but the parameter and result specification is to be supplied later or the name is a multiple entry point to a procedure. Bit 12 A value of one means that a MUTL name already exists for the procedure, but it does not have a specification, in this case bits 0-11 give the MUTL name, and calls to TL.PROC.PARAM and TL.PROC.RESULT will follow to give its specification. Bit 14 A value of one means the procedure (or subroutine) is an interface entity that is to be exported from the current module. Bit 15 A value of one means this is a specification of an assumed interface procedure (or subroutine) of another module.

	To reference a procedure from a library then its specification must be given. Bits 3 and 15 of NATURE are set to one to indicate this.

When compiling a library bits 3 and 14 are set to one to indicate that this procedure is in the libraries interface.

2) TL.PROC.PARAM(TYPE,DIMENSION)

This procedure defines the type of the next parameter in the current procedure declaration. A MUTL name is not allocated for the parameter by this procedure. Names for a procedures parameters are allocated by TL.PROC when it is called to commence the definition of the procedure.

Parameters:–

TYPE	TYPE SPECIFIER of parameter, see paragraph 23.3 for its encoding.
DIMENSION	0 scalar parameter >0 parameter is a vector containing the DIMENSION elements <–1 parameter is a vector its dimension is the value of literal whose MUTL name is DIMENSION negated.

3) TL.PROC.RESULT(TYPE)

This procedure defines the result type for the procedure (or subroutine).

Parameters:–

TYPE	Bit 0–13 contain a TYPE SPECIFIER of the result, see paragraph 23.3 for its encoding, in addition a zero value means the procedure has no result.

4) TL.PROC(MUTL.NAME)

Defines that a procedure (or subroutine) starts at the current position in the code segment. For procedures a name for each of its parameters as specified in the associated procedure specification is automatically allocated at this point.

Parameter:–

MUTL.NAME	MUTL name of procedure.

5) TL.PARAM.NAME(MUTL.NAME,[SYMB.NAME])

This procedure is called during the definition of a procedure to supply symbolic names to its parameters.

Parameters:–

MUTL.NAME	MUTL name of parameter.
[SYMB.NAME]	Bounded pointer to a byte vector containing the symbolic name of the parameter. It is used only for diagnostic purposes.

6) TL.PROC.KIND(KIND)

This procedure is called to specify the non–local accessibility of the current procedures labels and variables.

Parameters:–
KIND

bit 2	Specify non–local accessibility of the procedures variables.
1 -	Prohibited.
0 -	Allowed.

7) TL.END.PROC( )

Defines the end of the code for the procedure (or subroutine) most recently started.

8) TL.ENTRY(MUTL.NAME).

On calling this procedure the current position in the code is designated to be a multiple entry point for the enclosing procedure. Multiple entry is only allowed on static procedures, all entry points have an identical specification.

Parameters:–

NAME	MUTL name previously allocated by calling TL.PROC.SPEC with a NATURE of %2000, and this name is specified to call the procedure via the multiple entry point.

9) TL.BLOCK( )

This procedure is called to indicate the start of a block. No name is allocated for the block.

10) TL.END.BLOCK( )

This procedure is called to indicate the end of the block most recently started.

23.8. LABEL DECLARATION

Labels in code may be classified by their use and their origin, i.e. whether the label is source language or compiler generated. As most compiler generated labels are referenced in a very restricted manner, some MUTL implementations may be able to preserve actual machine registers across such labels. Labels are classified as follows.

i) Source language labels. Non–local (i.e. from textually embedded procedures) to these labels is not permitted. For local jumps to labels of this kind use the -> orders.

ii) Source language restart labels. These labels are always accessible non–locally from textually embedded procedure.

iii) Compiler generated labels.

As some labels need to be forward referenced, a specification of a label is always given before the definition of the value of the label. The specification and declaration of the label must appear in the same procedure.

1) TL.LABEL.SPEC([SYMB.NAME],LABEL.USE)

This procedure gives a specification for a label and allocates a MUTL name for the label.

Parameters:-

[SYMB.NAME]	Bounded pointer to a byte vector containing the characters of the labels symbolic name. It is used to identify the interface entity and for run time diagnostic purposes.
LABEL.USE	Bits 0 to 3. Values interpreted as 0 – Source language label. >1 – Compiler generated labels. Bit 14 = 1 Label to be exported as an interface entity. Bit 15 = 1 Specification of an assumed interface entity which is to be imported from another module.

2) TL.LABEL(MUTL.NAME)

This procedure declares a label value by assigning the current code address value to the label.

Parameters:-

MUTL.NAME	MUTL name of a label.

23.9. PICTURE DECLARATIONS

Code planting in MUTL caters for Cobol like editing operations. The editing is controlled by a picture specification.

1) TL.PIC(PICTURE)

This procedure allocates a MUTL name for a picture specification.

Parameter:

PICTURE	This is a bounded pointer to a byte vector containing a picture specification in a canonical form. Exact details of this encoding have not yet been resolved.

23.10. CODE PLANTING

Most instructions consist of a function and an operand as specified in the TL.PL procedure. Optimisation of loop control, when possible, is desirable for reasonable object code efficiency, therefore, MUTL also provides special code planting procedures for the most frequently used types of loops. MUTL distinguishes two special kind of loop.

a) Loops that are executed a number of times, where this number can be determined at the start of the loop, and no explicit control variable is required.

b) Loops that require a control variable, but the increment for the control variable is either +1 or -1.

For (a) the procedures TL.CYCLE and TL.REPEAT are called, for (b) the procedures TL.CV.CYCLE, TL.CV.LIMIT and TL.REPEAT are called. For other loops the appropriate sequence of TL.PL, TL.LABELS, etc. calls must be used.

1) TL.PL(FN,OP)

This procedure specifies one MUTL instruction. It may translate into several actual machine instructions or it may sometimes be optimised out. The latter is very often the case with the D instructions concerned with stepping through data structures. The general form of MUTL instructions is one-address, and they are in the main orthogonal in the sense that any function can be used with any operand. The exceptions to this rule are given in 23.10.3. A function (FN) specifies an operation and an operation type (or MUTL register) as shown in 23.10.1.

The operand part a MUTL instruction (OP) is in principle a MUTL name, but there are special encodings of this name to provide for literals, registers, stacked quantities etc, as described in 23.10.2.

23.10.1. Operation Codes

The FN parameter is encoded thus

```
bits 0 - 4 specify the operation
bits 5 - 6 specify the operation set
           0 indicates B operation
           1 indicates A operation
           2 indicates Organisational operation
           3 indicates D operation
bits 8 -15 for A and B operations, and ACONV, AMODE and ADDR= opcodes
            this field specifies the register number.
```

There is in effect a greater range of operation types than is apparent in the above since the A-register may be defined to be in any of the following basic modes

REAL(SIZE 32 or 64 or 128 BITS)
INTEGER(SIZE 8 or 16 or 32 or 64 or 128)
LOGICAL(SIZE 1 or 8 or 16 or 32 or 64 or 128)
DECIMAL(any size up to and including 80)
PTR(SIZE UNBOUNDED or BOUNDED)
TYPELESS(ANY SIZE or TYPE)

There are other modes for the A-register which cater for specific high level languages. These are known as extended A modes and at present these operations are only available on the A0 register. The extended modes are:

```
COMPLEX      -      FORTRAN
CHARACTER STRING OPERATIONS - FORTRAN and COBOL
```

Thus in the operation table given below, the A operations are given for each basic mode which is selectable.

			A FUNCTIONS						
	D	B	REAL	DEC	INT	LOG		PTR	ORG.FN
0	=>	=>	=>	=>	=>	=>	=>	=>	STK L
1	=REF	=-	=-		=-			=REF	STK PAR
2	=	=	=	=	=	=	=	=	ENTER
3	SEL FLD	-=			-=	-=			RETURN
4	SEL EL	&			&	&			
5	SEL ALT	V			V	V			ACONV
6	BASE	<<		SCALE		<<		BASE	AMODE=
7	LIMIT	>>				>>		LIMIT	STACK
8		+	+	+	+				STK LB
9		-	-	-	-				-> IF=
10		-:	-:	-:	-:				-> IF/=
11		*	*	*	*				-> IF>=
12		/	/	/	/				-> IF<
13		/:	/:	/:	/:				-> IF=<
14		==			==	==			-> IF>
15		COMP	COMP	COMP	COMP	COMP	COMP	COMP	SEG ->
16									ISEG ->
17									AECONV
18			MFN	MFN	MFN				AEMODE=
19		-=>			-=>	-=>			ADDR=
20		&>			&>	&>			
21		V>			V>	V>			
22									
23									
24		+>	+>	+>	+>				
25		->	->	->	->				
26		-:>	-:>	-:>	-:>				
27		*>	*>	*>	*>				
28		/>	/>	/>	/>				
29		/:>	/:>	/:>	/:>				
30		==>			==>	==>			
31		**	**		**				

The operators used in the above table have the following meanings.

=>	Store
=-	Load negative.
=	Load.
-=	Logical 'non-equivalence'.
&	Logical 'and'.
V	Logical 'or'.
==	Logical 'equivalence'.
<<	Logical left shift. The operand must be a positive integer.
>>	Logical right shift. The operand must be a positive integer.
+	Add.
-	Minus.
-:	Reverse minus.
*	Multiply.
**	Exponentiate.
/	Divide. For integer division this yields the nearest integer in the range zero to the mathematical result.

/:	Reverse divide.
COMP	The operand is compared with the register concerned and the T register set accordingly. The T register indicates one or more of the following six states namely, =, /=, >=, >, =< and <.
-=>	Logical 'non-equivalence' into store.
&>	Logical 'and' into store.
V>	Logical 'or' into store.
==>	Logical 'equivalence' into store.
+>	Add into store.
->	Subtract into store.
-:>	Reverse subtract into store.
*>	Multiply into store.
/>	Divide into store.
/:>	Reverse divide into store.
SCALE	Scales the decimal A register by a power of 10 specified by bits 0–7 of the operand which are interpreted as a signed integer. For a negative scale factor the digit in bits 8–11 of the operand is added to the last discarded digit in forming the result. Thus a digit of 5 in bits 8–11 rounds and a 0 truncates the result.
STK L	Stack link. Bits 0–11 of the operand contain the name of a procedure specification or zero. The latter value is for implementing interpretive procedure calls. Bits 12 and 13 given information about the procedure, a zero means that there is a definition associated with the specification, a one means entry is via a procedure variable without environment information, and a two means entry via a procedure variable with environment information. This opcode is required at the start of a procedure call sequence.
STK LB	Similar to STK L except the procedure being called is specified by it's FIND.N value which is contained in the current liter al, the operand is a literal that specifies the type of the result and is encoded as described in paragraph 23.3.
STK PAR	Stacks a parameter in a procedure call sequence. The operand must be a register whose mode is compatible with the parameter specification. For interpretive procedure calls and procedure called with STK LB the mode of the A register is taken as the parameter specification.
ENTER	Instruction used at the end of the procedure call sequence to enter the procedure as specified by the associated STK L. A zero operand enters the procedure associated with the specification given by STK L or STK LB, or it specifies the name of a procedure variable, or it specifies the A register in which case the A register contains the entry address of the procedure.
RETURN	Returns control from a called procedure to the instruction immediately following the ENTER used to call the procedure. The operand specifies the result for functional procedures. The result is passed back via the A register. An operand of %30dd means the A register contains the result and a zero operand either means there is no result.
ACONV	Sets a new basic mode and precision for the specified A register, and the contents of A are converted into the new mode as described in 23.10.4. The operand specifies the mode.
AMODE=	Sets the basic mode and precision of the specified A register at the start of an evaluation. The operand is an encoded mode as described in 23.10.4.
AECONV	Operates as ACONV but selects an extended mode for the A register.
AEMODE=	Operates as AMODE= but selects an extended mode for the A register.
STACK	Stacks the register specified by the operand. A stacked operand is removed by specifying an UNSTACK operand in the instruction.
->IF= ->IF< ->IF> ->IF/= ->IF=< ->IF>=	Transfers control to a label in the same MUTL segment if the status of T implies the condition of the instruction. Operand is a name of label type.
SEG ->	Transfers control to a label in the same MUTL segment. Operand is of label type.
I.SEG ->	Transfers control to a label in another MUTL segment. Operand is of label type.
=REF	Load a reference to the entity specified by the operand. The operand must be the MUTL name of a data item or D[ ].

SEL FLD	Selects a sub-field of the current field. The operand is a non-negative literal integer.
SEL EL	Selects the Bdd'th element of the current sub-field. The operand is a literal of the form %20dd where dd specifies a particular B register. A zero operand means select the B0'th element.
SEL ALT	If a field has multiple type definitions this selects the required type definition. A SEL ALT is not required if the first of the multiple type definitions is required. The operand is always a non-negative literal integer.
BASE	The B0 register specifies an element of a vector addressed by a pointer in the A or D register. This pointer is modified by this instruction so that the base of the vector now starts at the element specified. The operand specifies whether to maintain an unbounded or a bounded pointer (i.e. = 0 or 1).
LIMIT	The B0 register specifies an element of a vector addressed by a pointer in the D or A register. The pointer is modified so that the last element of the vector becomes the element specified. No operand is required.
MFN	Mathematical and built-in functions that operate on A registers. The operand is taken to be a function number and the available functions are given in 23.10.5.
ADDR=	Load a MUTL base register. The operand is a literal and it is encoded as follows:

```
Bits 0 - 7 MUTL Segment Number or Procedural textual level.
Bit 8      1 Load with MUTL segment address
           0 Load with stack frame address
Bit 9      0 Plant code to load base register
           1 Note that base register is loaded but do
           not plant code.
```

The operation codes for the extended modes are given in the table below

	COMPLEX	STRING
	COMP	CHAR
0	=>	L.STR
1	=-	R.STR
2	=	MOV
3	R=	E.MOV
4	I=	COMP
5		E.COMP
6		SRCH
7		E.SRCH
8	+	LEN
9	-	E.LEN
10	-:	ASC.COMP
11	*	E.ASC.COMP
12	/	LOAD
13	/:	EDIT
14		C.EDIT
15	COMP	N.EDIT
16		E.N.EDIT
17		MOV.B
18	MFN	
19		
20		
21		
22		
23		
24	+>	
25	->	
26	-:>	
27	*>	
28	/>	
29	/:>	
30		
31	**	

Where the operation codes mean

R=	Load real part only of complex accumulator.
I=	Load imaginary part only of complex accumulator.

The other operators are as for the basic modes except they operate on the complex accumulator.

Nine character string operations are provided:

MOVE	This involves concatenation of one or more strings (called right strings) to form a result string and storing this in a destination string (called a left string). If the result string is longer than the left string, then the string starting at the lefthand end of the result string of the same length as the left string is stored. If the result string is shorter, then the result string is transferred to the lefthand end of the left string and blanks inserted in the remaining part of the left string.
COMPARE	This compares two strings and sets the T status bits accordingly. The strings involved are a left and a right string,

	and each string consists of a concatenation of one or more strings. The left string is compared with the right string. If one of the strings is shorter than the other then it is extended to the right with blanks so that both strings are of the same length. Comparison proceeds by comparing the strings from left to right using the collating sequence of the character set.
SEARCH	This searches one string for the leftmost occurrence of another string. The left string is searched for the leftmost occurrence of the right string. If found its position, counting from one is put in the B register, otherwise B is set to zero.
LENGTH	This gives the length of a string in B. The left string is used to hold the string.
ASCII COMPARE	Similar to COMPARE but the comparison uses collating sequence described in ANSI, X3.4-1968 (ASCII).
LOAD	This loads a numeric character string, so that an immdiately following ACONV opcode enables conversion to a decimal representation in the A register.
EDIT	This performs editing as typified by an alphanumeric move in Cobol between two character strings. Editing is governed by a Picture specifier.
NUMERIC EDIT	This performs editing as typified by a numeric edited move in Cobol between a decimal value in the A register and a character string. Editing is governed by a Picture specifier.
MOVE BYTE	This moves the same literal value to every byte of a character string.

Each one of these operations is specified as a sequence of simpler character string instructions.

```
   Example

             E = F//G//H

   MUTL instructions

             MOV        :: Start of a MOVE operation
             L.STR E    :: Left string
             R.STR F    :: Result string of right strings
                           F, G and H
             R.STR G    :: Concatenated
             R.STR H    :: Together
             E.MOV      :: End of MOVE operation.
```

The character string opcodes provided are described below. A string operand is always a vector of byte sized elements.

L.STR	This opcode is used to specify the left string. If the character operation in which this is used permits concatenation of strings, then a sequence of L.STR functions specify that the left strings will be concatenated together and then act as a single string in the operation.
R.STR	As L.STR except it applies to the right string.
MOV	Start of a MOVE operation, this is followed by instructions to specify the left string and then the right string. The left string

	consists of a single operand, whereas the right string may consist of several concatenated operands.
E.MOV	This opcode terminates the specification of the right string and MOVE operation.
COMP	Start of a COMPARE operation, this is followed by functions to specify the left string and then the right string. Both the left and the right strings may consist of several concatenated operands.
E.COMP	This function terminates the specification of the right string and the COMPARE operation.
SRCH	Start of a SEARCH operation, this is followed by functions to specify the left and then the right string. Both of these strings may consist of several concatenated operands.
E.SRCH	This opcode terminates the specification of the right string and the SEARCH operation.
LEN	Start of a LENGTH operation, this is followed by instructions to specify the right string, which may consist of concatenated operands.
E.LEN	This opcode terminates the specification of the right string and the LENGTH operation.

The MOV, E.MOV, COMP, E.COMP, SRCH, E.SRCH, LEN and E.LEN opcodes do not have an operand. The stack may be used during an operation for holding temporary data.

ASC.COMP	As COMP but for ASCII collating sequence.
E.ASC.COMP	As COMP but for ASCII collating sequence.
LOAD	Loads a numeric character string.
EDIT	Start of an editing operation. The operand is a MUTL name of a Picture specifier. The destination for the edited move is defined by a single left string operand (i.e. L.STR) followed by the source which is defined by a single right string operand (i.e. R.STR).
E.EDIT	End of an editing operation.
N.EDIT	Start of a numeric editing operation. The operand is a MUTL name of a Picture specifier. The destination of the move is defined by a single left string (i.e. L.STR) followed by code to load the decimal A register.
E.N.EDIT	End of numeric editing operation.
MOV.B	This moves the current literal value to every byte of the character string specified by the operand.

23.10.2. Operands

For most functions the operand is one of the following:

1.	A zero which means the operand is a basic type literal having a value and type of the current basic type literal.
2.	A name of a previously declared item (1<OP<%800)
3.	A special operand (OP>%7FF).
%1002	V.SUB see TL.V.DECL in 23.5.
%1003	The operand is to be popped from the top of the current stack frame. MUTL requires that the use of the stack for temporary variables is determinable at translation time. This means that the corresponding STACK function must statically precede a %1003 operand, and that associated pairs of STACK functions and %1003 operands must be statically nested. As with most other operands of A and B opcodes this operand may be of a different type and size to the register, the operand type information is obtained from the corresponding STACK opcode.
%1004	D[] the item addressed by the D register.
%1005	specifies an operand that is normally used with the STACK and the D= instructions. This operand allows the stack to be used for temporary select variables (see 23.5). This operand with a STACK instruction pushes the current selection information of the D register onto the stack, and on the corresponding paired D= instruction with this operand, the stacked selection information is popped into the D register. Obviously several items of this type may be on the stack simultaneously, but as with select variables there are static scope implications. The STACK and D= with this operand may be statically nested, and the D= is paired with the immediately preceding unpaired STACK in the compiled code.

For the next six special operands the value of the operand for the instruction depends on the state of the T register.

%1008	a 1 if T implies =, otherwise 0
%1009	a 1 if T implies /=, otherwise 0
%100A	a 1 if T implies >=, otherwise 0
%100B	a 1 if T implies >, otherwise 0
%100C	a 1 if T implies =<, otherwise 0
%100D	a 1 if T implies <, otherwise 0
%20dd	a B register, bits 0-7 specify which one.
%30dd	an A register, bits 0-7 specify which one.

23.10.3. Restrictions on the Use of Operands

There are a number of exceptions to the general rule that any function can be combined with any operand.

a) Some functions have no operand as stated in the function description
b) Some functions have specially encoded operands as stated in the function description
c) Operands of B orders must be of type integer or logical
d) Operands of store and into store operation codes must match the size and type of the storing register.
e) For MUTL translators where there is only single A and B registers the B and A register operands are restricted to =, =>, STACK and STK.PAR opcodes.

23.10.4. Accumulator Mode Conversions

The operand of AMODE= and ACONV operations is taken as an encoded integer where the bits 0–13 give a TYPE SPECIFIER as in 23.3 to specify a scalar type mode; an array type generic mode is selected by a value of 2 in bits 0, 1 with bits 2–13 specifying the element type and the number of array elements specified by the current literal. In the case of ACONV bit 14 is also relevant and it specifies a KIND of conversion. Note that basic arithmetic modes have a value less 256 and have bits 0, 1 zero, while PTR modes have a value of 1 or 3 in bits 0, 1.

For the extended modes the operand of AEMODE= and AECONV operations is encoded as for AMODE= and ACONV, but the TYPE SPECIFIERs are as follows:

Complex	- TYPE SPECIFIER of an aggregate type containing 2 reals. Fortran always uses a TYPE SPECIFIER of %108.
Character string	- TYPE SPECIFIER of %83.

When the type given by the ACONV operand or the existing mode of the A register is unsigned integer then the precision of the A register is first changed to that of the new mode, and then the conversion is applied. This conversion changes the A register type and leaves in the register a binary equivalent value of its value in the old mode. Similarly when converting from an existing mode of unsigned integer, the precision of the A register is first changed to that of the new mode, and a type conversion, which leaves in the A register a binary equivalent value, is then applied.

The conversions available are given below

INTEGER TO REAL

INTEGER TO DECIMAL

REAL TO INTEGER

KIND = 0	Truncated conversion yielding the nearest integer value selected from the range zero to the floating point value.
KIND = 1	Rounded conversion yielding the nearest integer value to the floating point value.

REAL TO DECIMAL	
KIND = 0	Truncated conversion.
KIND = 1	Rounded conversion.
DECIMAL TO INTEGER	
DECIMAL TO REAL	
INTEGER TO INTEGER	Change in precision
REAL TO REAL	Change in precision
DECIMAL TO DECIMAL	Change in precision
INTEGER TO UNSIGNED INTEGER	
REAL TO UNSIGNED INTEGER	
DECIMAL TO UNSIGNED INTEGER	
UNSIGNED INTEGER TO UNSIGNED INTEGER	Change in precision
UNSIGNED INTEGER TO INTEGER	
UNSIGNED INTEGER TO REAL	
UNSIGNED INTEGER TO DECIMAL	
INTEGER TO PTR	The integer value becomes the address origin of a pointer.
PTR TO INTEGER	
KIND = 0	The address origin of the pointer becomes an integer value.
KIND = 1	The bound of the pointer becomes an integer value.
INTEGER/REAL TO COMPLEX	
KIND = 0	The integer/real accumulator value is converted and becomes the real part, and the imaginary part is set to zero.
KIND = 1	The integer/real value is converted and becomes the real part, and the imaginary part is undefined.
COMPLEX TO INTEGER	A truncated conversion is applied to the real part of the complex value to yield an integer value.
COMPLEX TO REAL	
KIND = 0	The conversion yields the real part of the complex value.
KIND = 1	The conversion yields the imaginary part of the complex value.
DECIMAL TO NUMERIC CHARACTER STRING	
	Conversion from decimal to a numeric character string, the result defines a right string operand as does the opcode R.STR. KIND specifies the sign representation in the string.
KIND = 0	Trailing sign.
KIND = 1	Leading sign.

KIND = 2 Trailing separate sign.

KIND = 3 Leading separate sign.

NUMERIC CHARACTER STRING TO DECIMAL
Conversion from a numeric character string to decimal, KIND specifies the sign representation as described above.

23.10.5. Mathematical Functions

MFN NO	REAL	INTEGER	DECIMAL	COMPLEX
0	ABS	ABS		ABS
1	MOD.1	MOD.1		CONJG
2	MOD.2	MOD.2		
3	SIGN.1	SIGN.1		
4	SIGN.2	SIGN.2		
5	DIM.1	DIM.1		
6	DIM.2	DIM.2		
7	MAX.S	MAX.S		
8	MAX.M	MAX.M		
9	MAX.F	MAX.F		
10	MIN.S	MIN.S		
11	MIN.M	MIN.M		
12	MIN.F	MIN.F		
13	SQRT	ODD		SQRT
14	SIN			SIN
15	COS			COS
16	LOG			LOG
17	EXP			EXP
18	LOG.10			
19	TAN			
20	ASIN			
21	ACOS			
22	ATAN			
23	ATAN.1			
24	ATAN.2			
25	SINH			
26	COSH			
27	TANH			
28	X SIGN			

Some of the functions (e.g. SIN,ABS) operate solely on the current contents of the accumulator. While others are used in a sequence (e.g. MOD.1,MOD.2) to provide a 'function' that operates on several values.

Example

For the Fortran statements

```
REAL X,Y,W,Z,V
Z = MAX (X,Y,Z,V)
```

The equivalent MUTL instructions are:

```
A.MODE = real
A = X
MAX.S
A.MODE = real
A = Y
MAX.M
A.MODE = real
A = Z
MAX.M
A.MODE = real
A = V
MAX.F
A => Z
```

In such a sequence the mode of the accumulator must be the same for each of the functions in the sequence.

The functions provided are as follows:

ABS	Yields absolute value. For complex mode the result is yielded in the A register whose mode and precision is that of the real part of the complex mode.
MOD.1 MOD.2	Yields a1 – int(a1/a2)*a2, where a1 is the value of the A register for MOD.1, and a2 is the value of A for MOD.2.
SIGN.1 SIGN.2	Yields |a1| if a2 >= 0, and –|a1| if a2 < 0.
DIM.1 DIM.2	Yields a1–a2 if a1 > a2, and 0 if a1 =< a2.
MAX.S MAX.M MAX.F	Yields the maximum of (as,am ..... am,af), where as is the value of the A register for MAX.S, am is the value for MAX.M, and af for MAX.F.
MIN.S MIN.M MIN.F	Yields the minimum of (as,am ..... am,af)
SQRT	Yields square root
SIN	Yields sine
COS	Yields cosine
LOG	Yields natural logarithm
EXP	Yields exponential

LOG10	Yields common logarithm
TAN	Yields tangent
ASIN	Yields arcsine
ACOS	Yields arccosine
ATAN	Yields arctangent
ATAN.1 ATAN.2	Yields arctangent of a1/a2
SINH	Yields hyperbolic sine
COSH	Yields hyperbolic cosine
TANH	Yields hyperbolic tangent
X SIGN	Extract sign. Yields −1 in A register if A value negative, otherwise yields 1 in A register.
ODD	Converts the A register into 1 bit mode and loads it with a value of one if a contains an odd integer, otherwise loads a zero.
CONJG	Yields the conjugate of the complex value.

The following groups of functions operate on a pair of values, each function is used once and in order.

```
MOD.1    MOD.2
SIGN.1   SIGN.2
DIM.1    DIM.2
ATAN.1   ATAN.2
```

The following groups of functions operate on two or more values. The function suffixed .S is used for the first value, the one suffixed .F for the last value, and the .M functions for all values inbetween.

```
MAX.S   MAX.M   MAX.F
MIN.S   MIN.M   MIN.F
```

Note that intermediate values in the above group functions are held temporarily on the stack, and that nested use of functions is permitted.

2) TL.D.TYPE(TYPE,DIMENSION)

The type of an operand referenced by the D register must be set by the compiler whenever a typeless data pointer is loaded into the D register. This procedure is called immediately before the call to TL.PL which loads a typeless data pointer into the D register.

Parameters:–

TYPE	TYPE SPECIFIER of operand addressed by D register, see 23.3 for its encoding.

DIMENSION	0 – Scalar addressed by D register. >0 – No. of elements in vector addressed by D register or length of bit string. <0 – D addresses a vector but number of elements unknown.

3) TL.CHECK(STATUS)

This procedure changes the level of program error checking. Note, a change in hardware error detection is dynamic in that it affects all subsequent instructions executed. Whereas the effect of a change in software error detection is static.

At the start of compilation all hardware checking is permitted, and all software checking is inhibited.

Parameters:–
STATUS

Bit 7 = 0(1)	means change level of software (hardware) checking
Bit 8 – 14	specify which of bits 0–6 are acted upon If bit (8+i) = 1 then act on bit i
Bit 0 = 0(1)	inhibit (permit) hardware array bounds checking
Bit 1 = 0(1)	inhibit (permit) overflow checking on B
Bit 2 = 0(1)	inhibit (permit) overflow checking on A in integer mode
Bit 3 = 0(1)	inhibit (permit) overflow checking on A in floating point mode
Bit 4 = 0(1)	inhibit (permit) overflow checking on A in decimal mode.

4) TL.RANGE(REGISTER,KIND,LOWER.LIMIT,UPPER.LIMIT)

This plants code to ensure that the value in the register specified by P1 is not outside the value range as specified by P3 and P4.

Parameters:–

REGISTER	%20dd a B register %30dd an A register dd specify which register.
KIND	specifies how P3 and P4 are encoded. Bit 0 – 0 P3 is an explicit literal -1 P3 is the MUTL name of a literal or a variable. Bit 1 – 0(1) Upper limit (not) defined. Bit 2 – 0 P4 is an explicit literal -1 P4 is the MUTL name of a literal or a variable.

5) TL.INSERT(BINARY)

The binary specified by the parameter is planted in the next available location in the current code area. Thus special operating system functions of an order code, such as dump and reload register, can be used from the systems implementation language MUSL.

Parameter:–

BINARY	Value of binary to be planted. The unit of binary planted is machine dependent.

6) TL.CYCLE(LIMIT)

This indicates the start of a loop which will be executed LIMIT times.

Parameter:

LIMIT	The MUTL name of an integer variable or an integer literal, note that for integer variables MUTL uses the variable specified in the limit test for the loop and does not take a copy of the variable. A value of zero means use the current literal. In addition the names %30dd and %1004 (see TL.PL) may be used.

7) TL.REPEAT( )

This defines the end of a loop.

8) TL.CV.CYCLE(CV,INIT,MODE)

This indicates the start of a loop which has an explicit control variable and a unit increment.

Parameters:

CV	The MUTL name of the integer control variable.
INIT	The initial value to be assigned to the control variable at the start of the loop. It can be the MUTL name of a variable or a literal, or zero meaning use the current literal, or %30dd or %1004 (see TL.PL).
MODE	
Bit 0,1,3 0	Increment is +1 and the loop is terminated when the control variable is greater or equal to the LIMIT specified in TL.CV.LIMIT.
1	As 0 except loop is terminated on a greater than condition.
2	Increment is -1 and the loop is terminated when the control variable is less than the LIMIT specified in TL.CV.LIMIT.
3	As 2 except loop is terminated on a less than or equal to condition.
4	(i.e. bit 3 = 1) Increment is +1 and loop is terminated when the control variable is equal to the LIMIT specified in TL.CV.LIMIT.
6	As 4 but increment is -1.
Bit 2=0	Control variable may be left undefined at the end of the loop.
Bit 2=1	Control variable is defined at the end of the loop.

9) TL.CV.LIMIT(LIMIT)

This is called after TL.CV.CYCLE, but before the start of the code for the loop body. This procedure specifies the limit value of the termination test of the loop.

Parameter:-

LIMIT	This can be the MUTL name of an integer variable of the same size as the control variable or an integer literal, or zero meaning use the current literal, or %30dd OR %1004 (see TL.PL).

10) TL.REG(REG.USE)

Once a register is loaded it is defined to be in current use. A register is no longer in current use after one of the following events:

EVENT	REGISTER
a) => and into store orders	B, A or D
b) COMP order	B, A or D
c) ST PAR with register operand	B,A or D
d) STACK with register operand	B,A or D
e) B or A used as a register operand when function is not =>	B or A
f) access of operand via D[ ]	D
g) SEL.EL	B
h) after a TL.MAKE function call	B
h) IF orders	T
i) BASE and LIMIT orders	B
j) TL.CYCLE, TL.CV.CYCLE, TL.CV.LIMIT with %30dd operand	A

or events a) to g) inclusive, the A and B registers may be retained in current use after the event by calling TL.REG immediately before the TL.PL call which causes such an event. The D register is only maintainable in current use after event a) when the operand is not a select variable and after f).

In other situations the compiler may need to inform the target code translator that a register is no longer in use.

Parameters:-

```
REG USE
 bit - 0 = 1 B    }
       1 = 1 A    }  retain register in current use
       2 = 1 D    }
       3 = 1 T    }

 bit - 4 = 1 B    }
       5 = 1 A    }  register no longer in current use
       7 = 1 ADDR }

bits 8 - 15 register number.
```

23.11. MUTL INITIALISATION

1) TL(MODE,FILE.NAME,DIRECTORY.SIZE)

Initialises MUTL for a compilation. This initialisation includes creating MUTL segment 0, this may be overridden where necessary by a call to TL.SEG to replace this default creation of MUTL segment 0. At the start of each module, area 1 of the module is automatically mapped to this code segment, and area 1 selected as the current code segment while area 0 (i.e., the stack) is selected as the current data area. The action of MUTL in this mode can be described more precisely in terms of MUTL procedure calls.

1) At TL time, MUTL takes the following action:–

   a) TL.SEG (0, Default Segment Size, −1, −1, 6).

2) At TL.MODULE time, MUTL takes the following action:–

   a) Map area 1 (code area) of the module to segment 0: TL.LOAD(0,1).
   b) Select area 1 as the current code area: TL.CODE.AREA(1).
   c) Select area 0 as the current data area: TL.DATA.AREA(0).

3) At TL.END time, MUTL plants in code segment 0 any code required for exiting from a program or library.

There are some restrictions when compiling in a one pass mode. In order to map data declarations as they occur then generally interface literals and anonymous types must be exported in the compilation run before they are imported into any other modules. Furthermore on some machines there may be some restrictions on the use of interface data variables.

When compiling a library the DIRECTORY.SIZE parameter specifies the maximum number of procedures expected in the library interface.

Parameters:–

MODE Bit 0 = 0	Produce necessary data structures to support run time diagnostics
Bit 0 = 1	Do not produce data structures
Bit 1 = 0	Normal compilation run
Bit 1 = 1	MUSS type compilation, note that this means that all on–stack declarations are within procedures.
Bit 2	Only used for a normal compilation run. A value of zero means compilation of all or part of a program, and a value of one means compiling all or part of a library.
Bit 3	Indicates action if any compilation faults are present. A zero value means abandon the compilation at the end of the first module encountered containing faults, a one means compilation continues even though faults are present, and in this case bit 4 specifies filing action.
Bit 4	A zero means that a faulty compilation yields no output files, and a one means output files are produced.
Bits 5–7	The use of this is machine dependent. For example when compiling MUSS control of the stack position may be required. See the MUTL implementation manual of the appropriate machine for details.
FILENAME	This gives the name of the file into which the output (i.e. code and data areas etc.) of a successful translation will be placed at the end of a translation (i.e. at TL.END).
DIRECTORY SIZE	
Bits 0–11	Specify the number of procedures in library interface.
Bit 15	A value of one means that MUTL will not create a library entry table.
Bits 12–14	Specify library organisation dependent information. See appropriate Library Interface Implementation Manual for details.

2) TL.END()

Terminates a compilation.

3) TL.MODULE()

Initialises the compilation of a module.

4) TL.END.MODULE(STATUS)

Terminates the compilation of a module.

Parameter:-

STATUS	A zero value means no faults detected by high level language compiler in this module, a value of one means faults.

5) TL.MODE(MODE,INFO)

This procedure allows the translation mode to be altered. The mode is set first by the MODE parameter of TL.MODE.

MODE	Bits 8-15 specify which bits of the translation mode to alter. If bit 8 is set then change bit 0, if bit 9 is set then change bit 1, etc. Bits 0- 7 specify the altered bit values of the translation mode. E.g. a mode of %8480 would change bit 3 to a zero and bit 7 to a one. The translation mode is interpreted as follows: Bit 0=1 Inhibit repositioning of fields in a type Bit 1=1 Inhibit code optimisation. Bit 2=0 Allign fields of a type to minimise store accessing. Bit 2=1 Inhibit above allignment thus reducing the size of some types.
INFO	This parameter permits further machine dependent control over the translation. See the implementation manual of the appropriate machine for details.

23.12. TARGET MACHINE PARAMETERS

It was mentioned earlier that in the interests of efficiency and reducing the complexity of a MUTL translator an optimising compiler is aware of certain parameters concerning the target machine being compiled for.

1) TL.ENQ(KIND)STATUS

This procedure yields information about the compiler target machine.

Parameters:

KIND	0 STATUS returns the MUTL type encoding for default length integers of the machine. 1 STATUS returns the MUTL type encoding for default length reals of the machine. 2 STATUS returns the number of procedural textual levels maintained by a display.

2) TL.ENQ.REG(MODE)[RESULT.VECTOR]

This procedure yields information concerning the number of MUTL registers available on the target machine. For a particular high level language MUTL allocates a fixed number of B registers and a fixed number of Base registers, but the mapping of A registers is not so straightforward. On many machines there are separate registers for different arithmetic modes, also a large register is often comprised of two or more smaller registers.

The RESULT VECTOR specifies the available registers.

The RESULT.VECTOR is encoded as

Word 0	Number of B registers
Word 1	Number of Base registers
Word 2	0
Word 3	Number of A register entries. Each entry consists of the three following words:
Word 0	Specifies which arithmetic modes that this set of machine registers support Bit 0 – 1 Real Bit 1 – 1 Integer/Logical
Word 1	Machine register size specified in bytes. If a register greater than this size is required then two or more consecutive registers are allocated as a single MUTL register. For example if a machine has four 32 bit registers, then the first two or the last two could be treated as a 64 bit register. Note that when several registers are needed, the number of registers is always a power of two (2{n}).
Word 2	Number of machine registers in this set.

The MUTL register A0 operates in all modes.

Thus an example RESULT.VECTOR encoding is

```
2          :: 2 MUTL B registers i.e. B0, B1
3          :: 3 MUTL Base registers i.e. BASE0, BASE1, BASE2
2          :: Machine has 2 kinds of arithmetic registers.
0
1          :: Set of Floating point registers.
8          :: 64 bit precision.
7          :: Seven registers of this kind numbered
           :: A1 to A7.
2          :: Set of 4 integer registers.
4          :: 32 bit precision.
4          :: Four registers numbered A8 to A11.
```

When several machine registers are treated as a single MUTL register the MUTL register number is that of the first register allocated, e.g. if A10 and A11 are treated as a 64 bit register, then A10 refers to the register pair, providing of course AMODE has been set appropriately. Furthermore compilers must choose consecutive registers such that they start on a register boundary within the register set which is an exact multiple of the number of machine registers needed for a particular A register. Thus A10 is a valid 64 register but A9 is not.

Parameters:

MODE	Some features of MUTL necessitate registers for their implementation, thus if such features are not needed in a particular high level language, such registers will, wherever possible, be made available as additional MUTL registers. 0 – MUSL 1 – FORTRAN 2 – PASCAL
[RESULT.VECTOR]	A bounded pointer to a vector of integers containing the Register Resource and Register Mapping tables.

23.13. DIAGNOSTICS

Run time diagnostics refer to items in terms of the source language program. Most of the necessary information to accomplish this is in the existing declarative procedures within MUTL, e.g. TL.S.DECL, TL.TYPE etc. Alternatively or in addition to high level language diagnostics a compile and data map of the compilation may be produced from the diagnostic information known at compile time. The procedures in this section exist solely to supply information to support diagnostics.

1) TL.LINE(LINE.NO)CODE.ADDRESS

MUTL notes the specified source language line number and this should be called for each line of source program that generates code.

TL.LINE yields the address of the next location in the code area to enable compilers to incorporate this information in a compile map.

Parameters:–

LINE.NO 0	means that the line number is obtained by the basic system IN.LINE procedure.
/=0	means that the parameter itself specifies the line number.

2) TL.BOUNDS(MUTL.NAME,[BOUNDS])

A vector variable in MUTL consists of a single dimension with a lower bound of zero. Languages that support multi–dimensional arrays, non–zero bounds or adjustable dimensions may elect (for reasons of efficiency) to map these arrays into a single dimension with a zero bound. TL.BOUNDS function is therefore to define this mapping, so that array diagnostics are in terms of the source language declaration of the array.

A multi–dimension array M may be mapped to a MUTL vector V in two ways. Consider an array of N dimensions with lower and upper bounds on each dimension of Li, Ui for i = 1, 2 .... N, and an array element (M(S1, S2 .... Sn). A 'Forward Mapping' is such that M(S1, S2 .... Sn) is mapped to

```
V((S1-L1)*D2*D3....*Dn + (S2-L2)*D3*D4....Dn
   + (Sn-1 - Ln-1)*Dn + (Sn-Ln)
```

A 'Reverse Mapping' is such that M(S1, S2 .... Sn) is mapped to

```
V(S1-L1+(S2-L2)*D1 + (S3-L3)*D2*D1
  ................ (Sn-Ln)*Dn-1*Dn*...*D1)
```

where Di is the size of the ith dimension i.e. (U1-L1+1).

Parameters:

MUTL.NAME	Bit 0-11 specify the MUTL name. A value of zero in bit 13 means the array has a Forward Mapping, and value of one means it has a Reverse Mapping.
[BOUNDS]	Bounded pointer to vector of integer elements. The bounds of a dimension are specified by a pair of elements. The first element specifies the lower bound and the second the upper bound of the dimension. Each element contains a MUTL name of a literal or a scalar variable. However a value of zero is permitted for U1 of a Forward mapped array, and for Un of a Reversed mapped array; this means that the bound is not known.

3) TL.PRINT(MODE)

This procedure controls the monitoring output of MUTL.

Parameter:-

MODE	This is bit encoded. Bit 0 = 1 means print name of all TL procedure calls and their parameters. Bit 1 = 1 means generate MUBL of all TL procedure calls. Bit 2 = 1 means print procedures map. Bit 3 = 1 means print code generated. Bit 5 = 1 means print data map. Bit 6, 7 used by implementors of MUTL for debug monitoring control. Object code in MUTL is produced from an intermediate form held in a vector. Setting bit 7 = 1 prints out this vector prior to code generation. See Section 27 of the MUTL Implementation Description for further details of the encoding of this intermediate form. Bit 8 = 1 means insert tracing instruction at the beginning of each line.

24. COMPILER WRITER UTILITIES

This section describes utilities to aid compiler construction and development.

24.1. DISASSEMBLER

1) OUT.PROG(I,I,I)

This procedure produces an assembler like listing of a portion of code as defined by the parameters. P1 is the segment number containing the code, P2 is the starting byte within this segment and P3 the finishing byte of the portion of code to be output. Output is sent to the currently selected stream. If P3 is zero then a suitable default is taken.

24.2. SYNTAB

SYNTAB is a Syntax Processing Package which allows a translator written largely in MUSL to have its syntax phase defined by a notation similar to that of BNF. First the package translates the syntax specification into MUSL then sends the resulting code to a file for compilation by the MUSL compiler. This MUSL code needs a SCAN procedure which may be called within the compiler. When entered this procedure attempts to match some text supplied in a vector parameter with the automatically generated vector containing the encoded form of the specified syntax. The SCAN procedure uses a top-down fastback technique.

24.2.1. The Notation for describing Syntax

The syntax specifications must appear at the beginning of the translator and be delimited by:-

```
BEGIN SYNTAX SPEC
and END
```

Thus a translator has the form:-

```
BEGIN SYNTAX SPEC
<SYNTAX SPEC>
END
<AUTOCODE PROGRAM>
```

There are five components in the SYNTAX SPEC the main two of which define the synes and cosynes.

```
<SYNTAX SPEC> =
<DELIMITER LIST>
<PROCEDURE LIST>
<COSYNE LIST>
<SYNE DEFINITIONS>
<COSYNE DEFINITIONS>
```

Syne definitions are syntactic rules of the languages written as in BNF except for these differences:-

1. Stylistic differences. The word SYNE must precede each definition and ::= is replaced by =.
2. Differences in ordering. The order of alternatives and of elements within alternatives are both different. Syne formulae are used by a left to right scanning algorithm which requires that:-

(a) Any alternative which is a stem of another comes after it.
(b) If one alternative is a special case of another it must come first.
(c) In recursive definitions there must be at least one leftmost element not recursive.

3. Metalinguistic Bracketing. Several alternatives may be specified as an element of another by enclosing them in square brackets.
4. Those syne definitions which are to be referenced by the autocode part of the compiler, (e.g., as parameters of the scan routine) must be preceded by a *. The result of this will be that a CONST declaration is generated to associate the syne name with the position allotted to it in a DATA VEC 'SYNES'.
5. Alternatives within syne definitions must be less than 128 elements long.

For example, a definition of <STATEMENTS> might take the form:-

```
SYNE <STATEMENTS> = <STATEMENT> [<STATEMENTS>|<NULL>]
```

where <NULL> indicates an empty string.

For both semantic and syntactic reasons it is convenient to be able to interrupt the scanning algorithm at defined points and execute code provided by the user. This is achieved by inserting an element into the syntax which is referred to as if it were a syne but is in fact defined as the name of a section of code (COSYNE) to be executed when the scanning algorithm reaches that point.

To make this facility more flexible the cosyne routine may have one numeric parameter the value of which is defined explicitly wherever the cosyne is used thus:-

```
<cosyne name (numeric parameter)>
```

Cosynes are defined as labelled sequences of Autocode instructions which operate within the SCAN procedure. Each sequence is labelled with the name of the cosyne it represents. Cosynes may use the variables and labels in the scan of which the following are the most useful:-

AS, WS, LINE, SYNTAX

Descriptors mapping the analysis record area, the working stack, the current statement and the syntax tables, respectively.

AP, WP, SS, SY

Pointers to the current position in AS, WS, LINE and SYNTAX, respectively.

TRUE, FALSE

Labels to which the cosynes may jump on completion.

In a translator it is usual to itemise the input string before applying the scan. This pre–scan pass may reduce delimiters to single pseudo symbols. A similar transformation has to be applied to delimiters appearing in the syntax. These delimiters and the value of the code to be assigned to them must be listed thus:–

```
        <DELIMITER LIST> =
                        DELIMITERS<NL>
                        <DELIMITER SPEC>
                        END<NL>
```

where <DELIMITER SPEC> = <INTEGER>/<STRING><NL>[<DELIMITER SPEC>|<NIL>]

<INTEGER>	is any integer in the range 0 – 255 and is the code to be assigned to the delimiter specified by,
<STRING>	which is any character string but all strings must begin with the same symbol, (e.g., 'for delimiters in quotes),
<NL>	represents a newline.

Each procedure name used as the parameter of a cosyne will be replaced by the integer which indexes its entry in the procedure list.

```
  Formally:-

           <PROCEDURE LIST> = PROCEDURE NAMES<NAME LIST><NL>
     where <NAME LIST> = <NAME>[,<NAME LIST>|<NIL>]
```

The COSYNE LIST contains the names of all the cosynes used in the syntax. Its form is:–

COSYNE NAMES<NAME LIST><NL>

FIGURE 1

Schematic Form of Input

```
BEGIN SYNTAX SPEC
DELIMITERS
128/'BEGIN'
.
.
END

PROCEDURE NAMES name, name, .....
COSYNE NAMES name, name, .....
SYNE<name>=---
*SYNE<name>=---
.
.
COSYNES
.
.
END SYNTAX SPEC
```

FIGURE 2

Schematic Form of Output
for a MUSL Compiler

```
LITERAL synename=index;
.
.
DATAVEC SYNTAX($L08)
.
.
END;
PSPEC SCAN(ADDR[$L08],ADDR[$IN],ADDR[$IN],ADDR[$IN],$IN);
PROCEDURE SCAN(SYNTAX,LINE,AS,WS,START);

->PASSWITCH;
LOOP2:
SWITCH SYNTAX[SY]\
cosyne name,
.
.
cosyne name;
.
PASSWITCH:;
.
**IN -1
```

24.2.2. The Syntactic Scan Procedure

The scan procedure has the specification:-

PROC SPEC SCAN (S, S, S, S, I32)

The parameters are:-

(i) The string SYNTAX which contains the complete syntax to be used in the scan and which is set up by the syntax processing phase. The scan maintains a modifier SY (I32 variable) at the current position in the syntax tables.

(ii) The vector LINE with 8-bit or 32-bit elements containing the text to be recognised. Each element is compared with a byte in the syntax tables. SS is maintains pointing to the last symbol recognised. On entry SS = 0.

(iii) or (iv) Two vectors AS and WS with 8-bit or 32-bit elements as the user requires. The modifiers within these vectors are AP and WP respectively, zeroed on entry to the scan. AS and WS are for use by the cosynes in generating an analysis record. AS or analysis space is conventionally the area in which the final analysis record is built up. WS or work space is the area used as temporary storage for intermediate levels of the analysis record. These areas are used for this purpose by the built-in cosynes VAL, NOVAL and IN. WS is used for output information as follows:-

WS[0] = -1 if the scan fails, WS[0] = 1 if it passes.

Consistent use of VAL will ensure that WS[1] is a modifier in AS pointing to the first element of the analysis record. The first free space in WS, usually WS[2] is the value of the modifier in LINE (SS) for the last symbol recognised. The analysis record created by the user starts as WS[1]. AS[0] contains the number of elements in the complete analysis record (i.e., if the last element is AS[4], AS[0] = 5).

(v) The address of the start of the SYNE to be scanned for. This is the I32 literal constant given by the name of a starred SYNE.

24.2.3. Operation of the Scan Procedure

24.2.3.1. Analysis Records

Obviously in BNF no record is kept of the way in which the source string fits the syne definitions. Also there is no specific way of recognising the end of an alternative. Both these functions are performed by cosynes. A terminal cosyne is required at the end of every alternative in a syne definition. This should generate the required record and end by transferring control to a specified point in the scanning routine, where the stack will be adjusted and the scan continued in the syne definition one level up. Two terminal cosynes are built into the system, they may be used as models for others. The first is UP which causes the scan to continue at the syne definition one level up and generates no analysis record. The other, VAL, generates an analysis record which as a tree structure form. The parameter of the reference to the VAL cosyne is entered as the first element of the current level of tree structure.

On some occasions it may be necessary to record information in the analysis record without creating a new level of tree structure. This is achieved by use of the cosyne IN, which inserts its parameter into the analysis record. Thus analysis records may be produced as in Figure 3. The pointers are stored as indices in the vector containing the analysis record.

```
SYNTAX:
   SYNE <SENTENCE> = THE[CAT<IN(1)>|DOG<IN(2)>]<VERB>THE WALL<VAL(0)>
   SYNE <VERB>     = SAT<PREP><VAL(1)>|RAN FROM <VAL(0)>
   SYNE <PREP>     = [ON<IN(0)>|UPON<IN(1)>]<UP>


INPUT:
                      THE DOG SAT ON THE WALL


ANALYSIS RECORD:
```

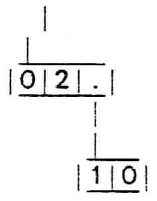


Figure 3. Analysis Records

24.2.4. Cosynes

These are labelled sections of code usually delimited by BEGIN and END. Several are built into the scan procedures and these are:-

BRANCH, SYMBOL, SYNE, MERGE, UP, ENDFALSE, ENDTRUE, VAL, NOVAL, IN

The VAL, UP and IN cosynes have already been discussed. The cosyne NOVAL acts as VAL but has no parameter and does not precede the analysis record level with a number. The other built in cosynes are of less interest to the user.

A reference to the UP cosyne occupies one 8-bit element of the internal encoding of the syntax; a syne reference occupies 3 elements and all other cosyne references occupy 2 elements.

The second of the 2 elements occupied by a cosyne holds the parameter and this may be accessed within the cosyne as:-

SYNTAX1[SY]

Cosynes may insert information onto the current level of analysis record by storing to:-

WS[WP] and advancing the index WP. They may also access the next symbol to be recognised as:-

LINE[SS + 1]

If a symbol is recognised thus, then the index SS should be advanced.

Cosynes should normally exit to one of the two built in labels TRUE or FALSE – according to whether recognition is to proceed or to backtrack in the syntax.

25. GRAPHICS UTILITIES

This Chapter describes the procedures provided in MUSS for handling graphics devices. The basic system distinguishes between two types of device: those that require bulk transfer facilities in order to communicate with the host computer ('indirect device access') and those that allow direct user access to the graphics store of the device ('direct device access').

With the latter type, the user is provided with the address of the graphics store when the device is assigned to the user. This address may subsequently be used for directly accessing the store. With the indirect type of device, commands are available for loading/unloading the graphics memory from the users virtual store.

25.1. COMMAND PROCEDURES

The commands associated with graphics management fall into two categories, namely the acquisition of the unit and the transfer of data. The acquisition commands are used for all device types in order to gain the sole use of a graphics unit. The data transfer commands are only applicable to the 'indirect' device accesses. In the following section individual commands within these two categories are described.

25.1.1. Organisational Commands

The first two commands described here are for the acquisition of a device and are used for all types of unit, the last two are used only for devices which require bulk transfer of data between host and the unit.

1) START.GR.OPERATION(I)

This procedure is used to gain access to a graphics unit. The required unit is specified in P1. The unit will not be assigned to the user process if it is being used by another user. If the unit is a direct access device, PW1 will return the starting address of the graphics store.

2) STOP.GR.OPERATION(I)

This procedure is used by the owner process to release the unit (P1) and make it available for re–assignation. It applies to both types of devices.

3) GR.READ(I,I,I,I)

This procedure is used to initiate a read operation. The actual operation is performed only if the unit is assigned to the calling process. P1 specifies the unit. P2 is the virtual address in bytes to which data is to be read. P3 states the maximum size of transfer in bytes. P4 gives the address in the graphics unit store from which data is to be read.

4) GR.WRITE(I,I,I,I)

This procedure is used to initiate a write operation. The actual operation is performed only if the unit is assigned to the calling process. P1 specifies the unit number. P2 is the virtual address in bytes of the start of data. P3 is the number of bytes to be transferred. P4 is the address in the graphics unit store to which data is to be written.

25.2. SPECIAL DEVICE FEATURES

25.2.1. Genisco

Genisco requires Direct Memory Access (DMA) channels to be set up for the operation. A typical session starts with a START.GR.OPERATION command and continues with GR.READ/GR.WRITE operations. The session ends with STOP.GR.OPERATION command.

While using GR.WRITE it must be noted that certain graphics store addresses (P4) have special meanings (see Genisco Manual). Below, the permitted values and their effects are described:

(i)	GENISCO address 0 is used to load the binary loader.
(ii)	GENISCO address %36 is used to start the device in execution mode.
(iii)	GENISCO address %50 is used to start the loading of the executive.
(iv)	A negative address is used to allow the GENISCO to continue from wherever it was halted.

Any other address will return an operation failed status.

APPENDIX 1 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 (XON)	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3 (XOFF)	33	3	53	S	73	s
14	DC4	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 separator)	3C	<	5C	\	7C	|
1D	GS (group separator)	3D	=	5D	]	7D	}
1E	RS (record separator)	3E	>	5E	↑	7E	‾
1F	US (unit separator)	3F	?	5F	–	7F	delete

APPENDIX 2 FAULT REASONS

Class 0 – Program errors

	Type
1	Arithmetic trap
2	Illegal instruction/operand
4	Disc transfer fails
8	Access violation
16	Segment undefined
32	Segment overflow
64	Stack overflow
128	Illegal organisational command call
256	Break point type 1
512	Break point type 2
1024	Program counter set to odd address
2048	Bound check fail
4096	Store limit exceeded

Class 1 – Limit violations

	Type
1	CPU runout

Class 2 – Timer runout

Class 3 – External interrupts

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)
8	Validate fail
9	Unable to map segment
11	Illegal channel number
13	No messages on specified channel
14	System unable to send message (no resources)

15	Channel closed
16	Faulty message sent to file manager
17	No response from destination process after send message and suspend.
20	System process number or process identifier invalid
21	Current process not supervisor
23	System unable to create a process (no resources)
25	Illegal username or password
30	Cannot create enough space for screen editor
32	Process name already 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
46	Illegal segment specified
47	File name already exists
50	Unable to update common segment
60	Graphics device unavailable
61	Graphics unit number out of range
62	Graphics boundary violation
63	Graphics operation failed
64	Graphics device to be used directly
65	Graphics transfer count is not positive
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
80	Subords limit reached
81	User name not unique
82	No more spare user available
83	User has used reusables
84	User name expected
85	Resource not available
86	Not authorised to payout
87	Parameter out of range
88	Current user not superior
89	Cannot erase default user

Class 4 – errors are initially returned to a process by a negative type code in PW0

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
FORTRAN 77 input/output errors	
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
130	Attempted use of unimplemented I/O feature

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
10	Parameter too large for EXP
11	Negative parameter with SQRT
12	Invalid exponentiation (**)
13	Invalid parameters with ATAN2
14	Negative parameter with LOG

15	Invalid parameter with ASIN
16	Invalid parameter with ACOS
111	FORTRAN Assigned GOTO fault. Label not in specified list, or not defined

Class 9	– System Library errors
10	Unable to open library
11	Maximum number of libraries already open
12	Unable to delete library
20	Unable to FLIP file

Index