UNIVERSITY OF EDINBURGH

Edinburgh Regional Computing Centre

EMAS User's Guide

A GUIDE TO THE USE OF THE EDINBURGH MULTI-ACCESS SYSTEM

December 1976

Edinburgh Regional Computing Centre

EMAS User's Guide

This Guide describes the Edinburgh Multi-Access System with particular reference to its appearance to the user.

PREFACE

This manual describes the Edinburgh Multi-Access System (EMAS), and in particular the user interface to the System - the Standard Subsystem. The first five chapters describe those parts of the whole System that support the Subsystem; the remaining chapters describe the Subsystem itself and the way in which it can be used. Appendices to the manual include a list of standard error messages, character code translation tables and a glossary of words and abbreviations used in EMAS user documentation.

It has been the policy of those responsible for running the System that its appearance should be developed to reflect the needs of its users. This process is continuing and as a result this manual will become out of date. It is unlikely that any future changes will invalidate information in the manual, but new facilities will be added and extensions made to the existing ones. The EMAS HELP command provides an up-to-date description of the facilities currently available.

One of the characteristics of EMAS is the ease with which users can add their own features to the standard facilities. Further they can readily make their additions available to their colleagues and to the wider user community. Apart from the facilities described in this manual there is a large amount of user-contributed material including commands, routines and packages. Information about what material is available can be sought from the Advisory Service, in the first instance.

The manual was typed by Mrs Anne Tweeddale and printed by the ERCC Reprographics Department. It is dedicated to all the members of the EMAS user community who by their suggestions, formal or otherwise, have created the present appearance of the System.

Roderick McLeod
December 1976

CONTENTS

CHAPTER 1
PAGING AND VIRTUAL MEMORY

This chapter contains a brief introduction to paging mechanisms and virtual memory. For two reasons it is felt necessary to explain these features of EMAS: first because they form a vital part of the system, and secondly because they are still comparatively unusual in large general-purpose operating systems.

Conventional storage allocation

All computers have some form of storage for programs that are currently being executed and for the operands they access. This storage, referred to here as main store, has to be allocated in some way. In a simple single-programming computer all the main store can be allocated to one program (figure 1). In a multi-programming computer main store has to be shared between more than one program according to the requirements of the programs. Even when the number of programs being multi-programmed is small there are problems:

* Available programs may not conveniently fit into store; for example, if the store has 500K bytes available for user programs and all user programs currently awaiting execution require 200K bytes then 100K bytes of store will be wasted.

* If programs have a wide variety of storage requirements the main store may quickly become fragmented; for example, if in figure 2 program A terminates and the only available programs to run require 250K bytes, one will be loaded as in figure 3 leaving a gap of 50K bytes at the top of the store. If program B now terminates there will be a total of 250K bytes of store free, but a single program requiring 250K bytes will not be able to run because the store which is free is not in one contiguous area.

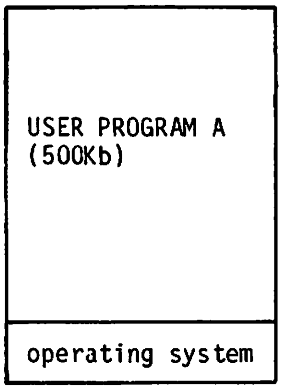


Figure 1

All available store allocated to one user program

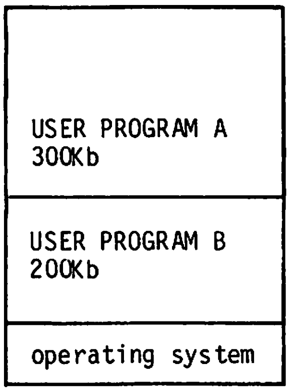


Figure 2

Available store allocated to two user programs

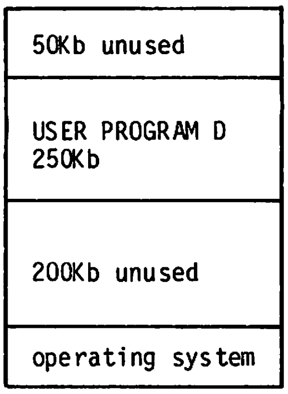


Figure 3

A second 250Kb job cannot run because store is fragmented

Further problems in a multi-access environment

Further problems of storage allocation arise in the case of a multi-access system:

* Whereas in a multi-programming batch system it is normal to allow typically 4 programs to share the available resources at any one time, in the case of a multi-access system 40 or more users may wish to use the computer simultaneously.

* The division of computing into discrete jobs of fixed size, which is feasible in a multi-programming service, is not possible in a general purpose multi-access system

where, for example, one user may be editing a program one minute, requiring very little store, and compiling it a few minutes later, requiring a large amount of store.

* As well as making varied demands for store each user will make varied demands for computing. Thus, whereas in a batch system each job normally runs to completion and is only delayed by waiting for peripheral transfers (e.g. blocks to be read from tape), in a multi-access system a user may take several minutes considering a reply to a prompt at his terminal. During this time no computing is being done for him and any store allocated to him is wasted.

Thus storage allocation in a multi-access system is concerned with many processes, each of varying size and making intermittent use of the central processor.

PAGING

To allocate storage effectively EMAS uses paging. This involves some special hardware and part of the code in the supervisor (see chapter 2). The main store of the 4-75 can be divided into pages of 4096 bytes each. Pages of store can be allocated to a program in any part of the store, wherever they happen to be available. The purpose of the paging mechanism is to modify the addresses of the pages so that to the running program they appear to be contiguous. Thus a simple user program might use four pages which it addresses as 0, 1, 2 and 3, although these pages might in fact be located at pages 8, 3, 14, and 1 in the main store (see figure 4).

The name 'virtual store' is given to the effective store addressed by the program.

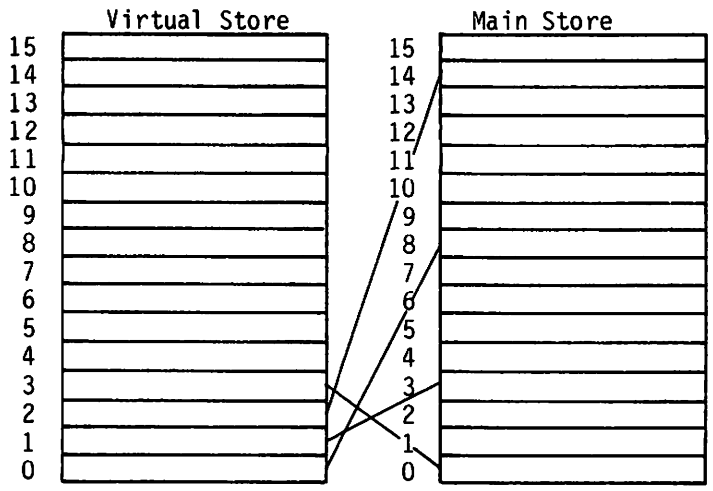


Figure 4 Example of Paging

One of the problems not considered so far is the great difference between the total storage requirement of all 50 processes and the size of the main store. EMAS has typically 200 pages of main store available to user processes. During any period of a few minutes user processes may access more than 1000 different pages. The pages not in main store are held on backing store and are brought into main store as required. Each page used by a user process must have a location on the backing store to which it can be returned when its space in main store is required by another process. The backing store consists of a discfile for the storage of fairly permanent copies of material, and a number of fast drum stores which hold, usually for short periods, transient copies of recently accessed pages.

Page faulting and virtual memory

An important characteristic of paging is the ability to deal with the varying demands of a program for storage. If a program starts to access an array which it has not accessed for several seconds the page containing the array may have been moved from main store to the backing store. As a result a 'page fault' occurs and the program is held up while the supervisor locates the page and reads it into main store. The program is then allowed to continue. In almost all cases the user knows nothing about this and need take no special action. A second effect of paging is that it allows any user to have a virtual memory (i.e. an addressable space) far larger than the real main store. In the case of EMAS each user can address an apparently contiguous area of 13 Mbytes.

The effect of paging on user programs

One of the attributes of paging is that its operation is transparent to the user program. The program behaves as if it were running in a normal main store and most users do not have to take any special steps. The large virtual memory makes it possible to run far larger programs than would be the case if paging were not used, without having to resort to overlaying. Furthermore, programs which would otherwise manipulate work files on magnetic tape or disc can use large areas of work space in the virtual store.

Mapping files

One fundamental difference between EMAS and many other operating systems concerns file handling. Whereas in a conventional system files are read, block by block, into buffers in main store and then moved to the user's area, in the case of EMAS files are 'connected' in the virtual memory. This activity, described more fully in Chapter 3, involves locating a 'hole' in the virtual memory and mapping the file pages on the disc into that hole. The effect is that the file can be accessed as if it were a part of the main store. It is important to understand that there is thus no distinction, to the system, between data files and program files and even work areas such as the stack, used by IMP programs to hold variables and arrays. Each file has an address in the virtual memory and consists of one or more pages, of which there is always a copy on the discfile; there may also be a copy on a drum, and when being used there will be a copy of some of its pages in the main store. An address in virtual memory is referred to as a 'virtual address'.

As explained in chapter 11, file handling routines have been made available which simulate those normally provided, in order to simplify the transfer of programs to and from EMAS. In addition however, there are user-level routines to enable users to map files onto arrays in their programs. These routines make possible efficient and powerful data manipulation with a minimum of code (see chapter 13).

Limiting page turns

In one important respect the use of a paged store is different from the use of a conventional store. Whereas in a conventional store variables can be accessed in a random order with no variations in the time required to access each one, in a paged store there will be a page fault each time a variable on a new page is accessed. This page fault will not affect the results of the program but will affect the time taken to complete the program. In the case of programs which require a total of less than 1/4 Mbyte for the whole program and data area this is unlikely to be important. The programs that are affected are those which access, for example, elements of large matrices in a random order. Such programs are likely to have elapsed times far longer than programs which use equivalent amounts of central processor time but which do not access a large number of different pages. This is particularly relevant for large programs that are to be run in foreground mode, since the elapsed time is then the time the user will have to sit at a terminal awaiting the completion of his program.

Further information

This chapter contains only a simplified description of paging. Further details of the paging hardware and software are contained in references 4, 5 and 6.

CHAPTER 2
INTRODUCTION TO EMAS

EMAS is a large, general purpose, multi-access operating system run on two ICL 4-75 computers at ERCC. Its important characteristics are:

* Multi-user - up to 120 users in theory, and typically 90 in practice, can access the dual machine system simultaneously.

* Interactive - its primary use is from interactive terminals, although batch computing is also supported.

* Paged - store is accessed via paging, resulting in each active user being able to have a large virtual memory (see chapter 1 for a description of paging).

The system is divided into two levels (figure 5):

* The resident supervisor - which remains in the main store all the time.

* The virtual processor - of which there is one for each active user or systems process.

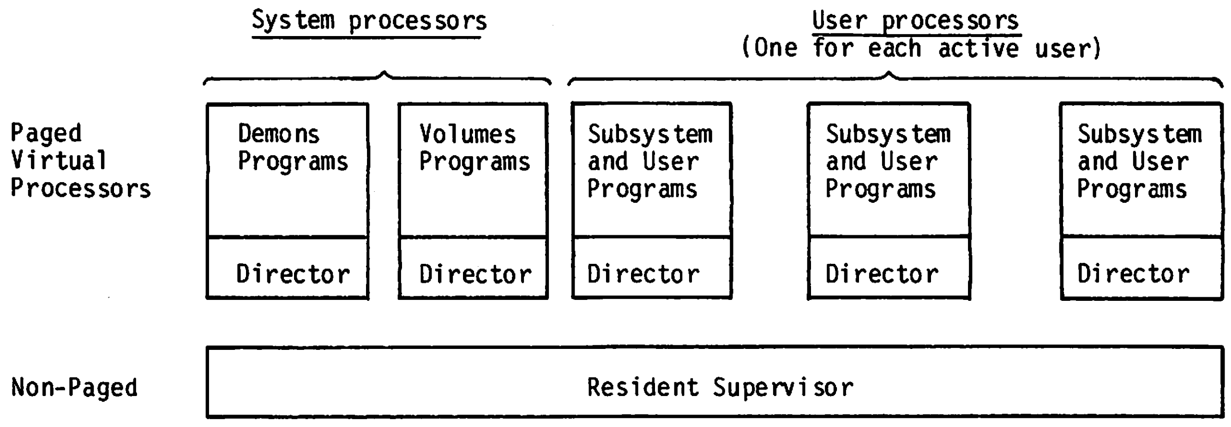


Figure 5 Simple Overview of EMAS

THE RESIDENT SUPERVISOR

The resident supervisor is permanently in main store; that is, it is not paged. It is responsible for:

* allocation of resources between virtual processors

* scheduling virtual processors

* moving pages to and from backing store

* controlling all peripherals

VIRTUAL PROCESSORS

Each active user has a virtual processor. This consists of a virtual memory containing two levels of software (see figure 6):

* a paged supervisor (called the director)

* a Subsystem and usually some user programs

The director maintains a table of information about the contents of the virtual memory; this is used to locate pages on the backing store. Whilst it is running the director can access the whole virtual memory, but when it hands over control to the Subsystem or to a user program then part of this table of information is cleared out, to prevent access to the director code and tables. This mechanism ensures that faults in the Subsystem or in user programs cannot result in corruption of director tables. Since these tables include information about, for example, other users' files, this mechanism is essential to preserve the security of the system. There is no hardware protection, however, between user programs and the Subsystem. This means that both unintentional and intentional corruption of Subsystem tables is possible. This can result in corruption of the user's own files and, at times, in his process terminating completely. What is important however is that no matter how serious the corruption in his own virtual memory, it cannot affect other users of the system.

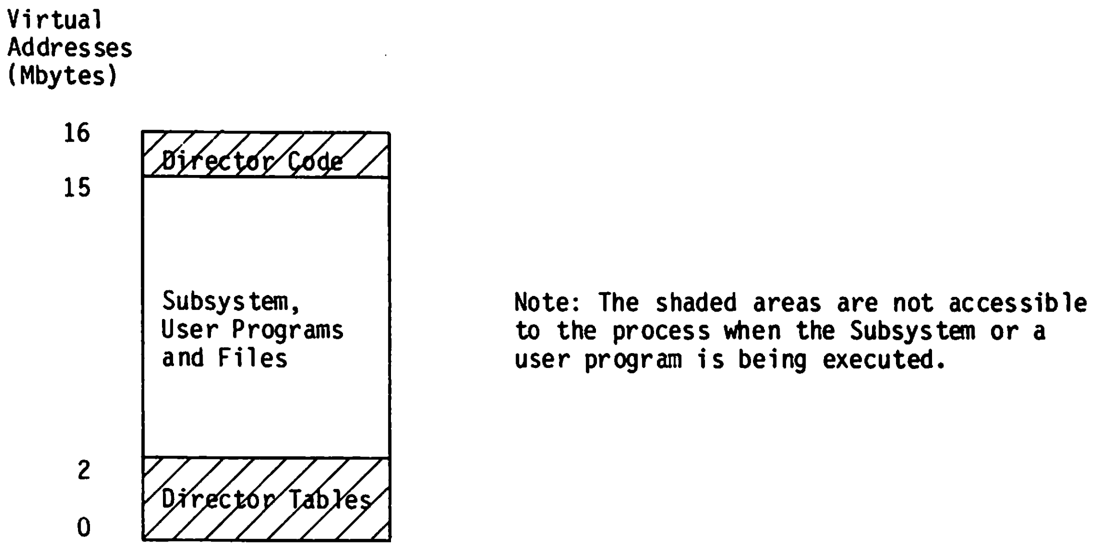

Virtual
Addresses
(Mbytes)
16
15
2
0
Director Code
Subsystem,
User Programs
and Files
Director Tables
Note: The shaded areas are not accessible to the process when the Subsystem or a user program is being executed.

Figure 6 Layout of the Virtual Memory of a Virtual Processor

The director

Apart from maintaining the table of information about files connected in the virtual memory, the director also performs the following tasks:

* organisation of the immediate file store (see chapter 3)

* communication with system processors (see below)

* communication with an interactive terminal (see chapter 5)

* dealing with failures in the Subsystem or user program such as 'Time Exceeded' and 'Divide Error'

System processors

Apart from virtual processors occupied by users there are a number of system functions that run in their own virtual processors. In many ways they are similar to user processors but they have some special attributes:

* they are privileged - that is, they can access information and obtain services from the resident supervisor which are not available to user processors

* they normally run without any interactive terminal

The two most important system processors are the demons processor and the volumes processor. The demons processor is responsible for

* handling input files from slow devices, e.g. card readers (see chapter 4)

* handling the output of files to line printers etc. (see chapter 4)

* scheduling batch jobs (see chapter 19)

* the verification of names and passwords at log on (see chapter 5)

* controlling communications to remote terminals (see chapter 4)

The volumes processor is mainly concerned with the organisation of the archive store (see chapter 3).

Scheduler

Part of the supervisor, known as the scheduler, is concerned with sharing available main store and central processor time between the various processors competing for these resources. Since EMAS is primarily intended as a multi-access system for interactive computing, the scheduler is designed to give priority to processes which make comparatively small demands for store and processor time. Put another way, the scheduler gives a lower priority to jobs which require large amounts of either resource. Thus processes which display characteristics of batch jobs, or are in fact started as batch jobs, only get resources when other processes are not waiting for them; see also reference 4.

HARDWARE CONFIGURATION

This manual does not contain a detailed description of the hardware configuration, or of any part of the hardware. The overall structure is summarised in figure 7.

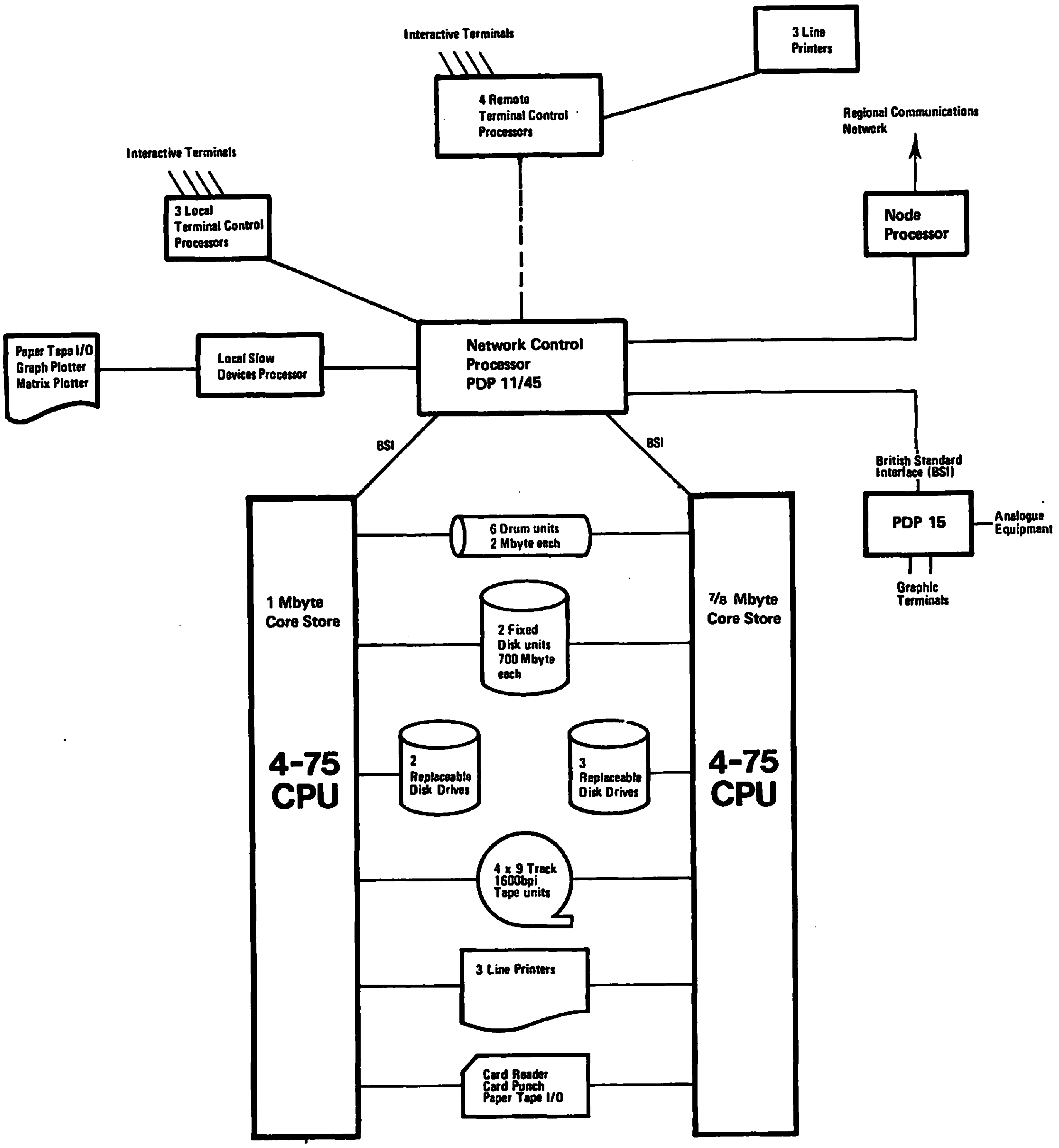

Interactive Terminals
3 Line Printers
4 Remote Terminal Control Processors
Regional Communications Network
Interactive Terminals
3 Local Terminal Control Processors
Node Processor
Paper Tape I/O Graph Plotter Matrix Plotter
Local Slow Devices Processor
Network Control Processor PDP 11/45
BSI
BSI
British Standard Interface (BSI)
PDP 15
Analogue Equipment
Graphic Terminals
6 Drum units 2 Mbyte each
1 Mbyte Core Store
2 Fixed Disk units 700 Mbyte each
7/8 Mbyte Core Store
4-75 CPU
2 Replaceable Disk Drives
3 Replaceable Disk Drives
4-75 CPU
4 x 9 Track 1600bpi Tape units
3 Line Printers
Card Reader Card Punch Paper Tape I/O

Figure 7 Simplified Diagram of EMAS Hardware

The System runs on two ICL 4-75 computers each having associated with it a fixed discfile with a capacity of about 700 Mbytes. The communications with interactive terminals and remote medium speed terminals (card readers and line printers) is via a Network Control Processor (a DEC PDP11/45 computer) and a series of Terminal Control Processors (mainly DEC PDP11/10 computers). The Network Control Processor is connected via fast links with both of the 4-75s, which means that any terminal can provide access to either.

Apart from the items mentioned above, there are

* Six drums between the two 4-75s, normally configured as three on each machine.

* Four 9 track magnetic tape decks, used almost exclusively for archive and back-up storage purposes (see chapter 3).

* Five replaceable disc units, used exclusively by the system.

* Various slow peripherals connected either to the 4-75s or the Network Control Processor. The precise configuration can vary, but provides paper tape and card input and output, and line printer output.

References 6 and 7 contain full details of the ICL hardware.

The whole configuration is interconnected in such a way that it is possible to run an EMAS service for all users, albeit degraded, in the event of a failure of almost any component, including a central processor, but excluding the file store.

Interactive graphics

In addition to the system described above there is a Digital PDP15 computer connected via fast links to the two 4-75s. This is used to provide, amongst other things, interactive graphics facilities on EMAS.

System integration

To a large extent EMAS can be seen by the user as one system. Normally he does not need to know that it runs on a pair of computers. The architecture of the 4-75 precludes complete integration, although this limitation has been partly overcome by use of the Network Control Processor. Each user is assigned to a particular machine and during normal service he can only use the system if there is room on his machine. Further he is not able to write to files belonging to users on the other machine even if they have given appropriate access permission; in practice this is not a serious restriction. On the other hand he can log on to either machine from any interactive terminal connected to an EMAS Terminal Control Processor.

CHAPTER 3
THE FILE SYSTEM

This chapter describes the EMAS file system. Files are used in EMAS for a wide variety of purposes. They can contain programs and data as in conventional systems but also they are used to hold temporary information such as the variables and arrays for a running program. The file system can be divided into two parts: immediate files, which are held on the discfile and are available for use whenever the user logs on to the system; and archive files, which are held on magnetic tape and which have to be transferred to immediate storage before they can be used.

All files are made up of one or more pages of information, the size of each page being 4096 bytes. As explained in chapter 1, when they are being accessed these pages are mapped onto pages in the user's virtual memory, and access to any particular part of the file is then achieved by addressing the apropriate part of his virtual memory. One way in which EMAS differs from some other systems is that it frees the user from any concern about the physical layout of his file. The system controls the way in which a file is stored on disc or on tape.

IMMEDIATE FILE STORE

The immediate file store is organised by the director. The director is only concerned with files as sequences of pages; it is not concerned with the internal structure or contents of files. The effect of this is that a number of Subsystem functions - those which are passed straight on to the director - are not concerned with file types; for example those functions initiated by the commands DESTROY and RENAME.

Naming files

Each file in immediate storage has a unique name which comprises two parts - the name of its owner (the 'ownername') and a file name given by the owner to distinguish it from his other files (the 'filename'). The two fields are separated by a full stop. Files created by users should have filenames of between 1 and 8 characters which should all be upper case letters or numerals, the first character being a letter. Examples of valid full filenames are:

```
ERCC06.FILEABC1
ERCC99.N
LMPT01.FIRE
```

The filenames of files created by the Subsystem usually start with the characters 'SS#' to avoid conflict with files created by the user. For example:

```
ERCC06.SS#STK
ERCC99.SS#LIST
```

Note that when referring to his own files a user does not normally have to prefix the filename with his ownername. Thus user LMPT01 would refer to the file in the example above as FIRE. When referring to files belonging to other users, however, he must always use their full names.

Security of files

The information contained in a file can only be read or altered if the file is connected in the user's virtual memory. The Subsystem can only cause a file to be connected by making a call on the director, and the director will only connect a file if appropriate access permission to the file has been given by its owner (see below). There is thus no way in which a fault in the Subsystem or in a user's program can enable him to access or corrupt information not intended for him. This arrangement makes it possible to store confidential information in EMAS without risk of corruption or illegal access.

Access permission

Each file has associated with it access permission information. Four different types of access are possible:

READ (unshared)
WRITE (unshared)
READ SHARED
WRITE SHARED

Any or all or none of these modes can be given to the file in respect of

* the owner
* all other users
* a specific user
* a group of users

When created, a file has all modes of access permitted to its owner and none to anyone else. Thus, unless a user explicitly permits a file to someone else, it is only available to himself.

The Subsystem command PERMITFILE is used to set access permission information for a file. Its use is described in chapter 7. When determining what access permission is allowed to a user who wishes to connect a file, the director follows these rules:

* If connecting a file belonging to self then use the access permission explicitly for self.

* If connecting a file belonging to another user then use the first of the following which applies:

  1. If specific permission has been granted to this user then use it.
  2. If group permission has been granted to a group that contains this user then use it.
  3. Use the 'everybody else' permission.

Note that if this user is included in more than one group of users the resulting permission is undefined.

Connect modes

A file can only be connected in a particular mode in a user's virtual memory if both the following conditions are satisfied:

* It is permitted to the user in the required mode.

* It is not connected in another virtual memory in a conflicting mode.

The following connect modes can be used:

* READ (unshared) - the file can only be connected in this mode if it is not connected in any other virtual memory. Whilst so connected it cannot be connected in any other virtual memory, even if permitted. Whilst connected in this mode it cannot be written to.

* WRITE (unshared) - all the attributes of READ (unshared) apply except that it can be written to or modified.

* READ SHARED - the file can be connected in a virtual memory in this mode if it is connected in one or more other virtual memories in the same mode or if it is not connected in any other virtual memory. Whilst so connected it can be connected in further virtual memories but only in READ SHARED mode. It cannot be written to or modified.

* WRITE SHARED - the file can be connected in a virtual memory in this mode if it is connected in one or more other virtual memories in WRITE SHARED mode or is not connected at all. Whilst so connected it can be connected in further virtual memories in WRITE SHARED mode. It can be written to or modified.

The above information is summarised in the following table:

Connect Mode	Whether allowed if already connected in another virtual memory	Can be read from or executed	Can be written to or altered
READ	Never	Yes	No
WRITE	Never	Yes	Yes
READ SHARED	Yes if connected in Read Shared Mode	Yes	No
WRITE SHARED	Yes if connected in Write Shared Mode	Yes	Yes

Notes

* WRITE and WRITE SHARED modes allow both writing and reading.
* There is no separate access mode for executing a program. If a file is permitted to a user for reading and it contains a compiled program then it can be executed by him.

The mode chosen for access by the Subsystem, when necessary, depends on:

* the use to which the file is being put; e.g. if an attempt is made to EDIT a file then it will have to be connected in a WRITE mode
* the access permissions allowed to the user
* the connect mode, if any, of the file in other virtual memories

If a suitable mode cannot be used the file is not connected and a failure occurs. Further details are given in the chapters describing the Subsystem facilities relating to files.

The structure of the file system

The immediate file store is divided into a number of separate parts, each of which is stored on a discrete part of the discfile. One result of this is that the effect of a discfile fault can often be confined to the users who have files on the faulty part.

This modularity of the file system is not of great significance to the user and normally he does not need to know which part of the file system contains his files. As explained in chapter 2, however, EMAS users are divided between two machines and at times it is necessary to know which files are on which machine. Subject to access permission being granted (see above) a user can access any files on the same machine as his own in any mode. Access to files on the 'other' machine is restricted to READ and READ SHARED modes only. In practice this should not be much of a restriction since most files are permitted to other users only for reading. It is, however, possible to transfer a file between users on different machines using the commands OFFER and ACCEPT (see chapter 7).

The user file index

Each part of the file store has a directory which is divided into a number of separate user file indexes. Each user's file index contains three parts:

* System File Information (SFI) - this contains information about the user's process which has to be retained between sessions, such as the size of his STACK (chapter 10) file, the OPTIONS (chapter 21) he has selected and the USERLIB (chapter 10) he has selected.

* File descriptors - there are 32, 64, or 128 of these, depending on the size of the index. One is required for each file owned by the user in the immediate store. These files include those created explicitly by the user and those created on his behalf by the Subsystem. Each file descriptor in use contains, for one file:

  * its name
  * its size
  * its access permissions to its owner and to 'everyone else'
  * its current connect mode
  * the number of virtual memories in which it is connected
  * its CHERISH status (see below)
  * usage information about it (see archive storage, below)

* List cells - this area, which contains a linked list of cells (32, 96 or 224 depending on the size of the index), is used to contain additional information about some of the user's files. Cells are used thus:

  * for each file currently on OFFER to another user (see below): 1 cell
  * for each file larger than 1 segment (16 pages): 1 cell for each segment
  * for each file permitted to a specific user or user group: 2 cells for each separate permission

Information about a user's file index in respect of a particular file or of all his files can be obtained from various Subsystem commands; see chapters 7 and 8.

File creation and extension

Many Subsystem functions make calls on the director to create files. File creation will fail if:

* the user's file index is full
* the part of the file system containing the user's index is full
* the user already has a file of the same name
* an attempt is made to create a file in another user's file index

The director imposes a limit of 1024 pages (4 Mbytes) on each file. At the time of writing a lower limit is applied by the Subsystem because of the limited amount of immediate storage available.

The size of an existing file can be altered, to a value between 1 and 1024 pages. This function is requested by the Subsystem, as required.

Transfer of ownership of a file

The Subsystem commands OFFER and ACCEPT result in a file being transferred from one user to another. The information relating to the name, the size, the location on the discfile and the CHERISH status is transferred from one file index to another. Note that all other information normally held in the file index (usage information and access permissions) is not transferred. Note also that the OFFER/ACCEPT mechanism can be used between users on the same or different machines.

Back-up of immediate file store

In order to protect users' files from hardware and system faults, they are copied onto magnetic tape once each day ('backed up'), subject to the following rules:

* A file must be marked for this purpose by use of the CHERISH command. Note that files created from card or paper tape input are automatically CHERISHed but otherwise files are created without the CHERISH marker being set. Note also that un-CHERISHed files will be DESTROYed by the system after a period (currently 4 weeks) of non-use.

* A file will only be backed up if it has been altered since the last time it was backed up. (Strictly, it is sufficient to have connected the file in WRITE or WRITE SHARED mode to cause back-up to take place.)

If, because of a failure, it is necessary to recover users' files from back-up tapes, the files are replaced on the discfile with the CHERISH status and any access permissions in force at the time they were backed up. Note the following points:

* Any alterations made to the file between the time of the latest back-up and the system failure will be lost.

* Files which have recently been DESTROYed may be copied back to the immediate file store. It will thus be necessary for the user to DESTROY them again.

THE ARCHIVE FILE STORE

The Archive File Store is held on magnetic tape, quite separately from the back-up tapes. Files are moved from the immediate store to archive store in the following circumstances:

* as a direct consequence of using the command ARCHIVE in respect of the file

* indirectly as a result of CHERISHing a file but not accessing it for a period (currently 4 weeks)

The command ARCHIVE and related commands for restoring archived files to immediate store are described in chapter 7. Unused files are moved onto archive storage in order to free the limited space on immediate store for material which is being actively used.

There is currently no limit to the number of files a user can have on archive store, nor to the length of time they are left there. On the other hand users are encouraged to tidy up their archive file list periodically.

Unlike files on immediate store it is possible to have two files on archive with the same name. In such cases the date of archiving is used to indicate which file is to be restored.

CHAPTER 4
INPUT AND OUTPUT OF FILES

The Input and Output of files to and from EMAS is controlled by the DEMONS systems process (see chapter 2). It is not possible for users to access directly file input and output devices, e.g. card readers or line printers. Instead they can instruct the DEMONS process to carry out operations on these devices with reference to particular files.

INPUT

It is possible to read files into immediate storage from cards or paper tape. This is usually achieved by submitting the file, with appropriate Job Control Language (JCL) statements, to be read on a card reader or paper tape reader attached directly to EMAS. Alternatively, input can be sent from terminals connected to the medium speed network; files can also be transferred from other processors connected to the network. The details of these operations are subject to changes as the network is developed; the user is referred to the current HELP information.

Card reader input

Cards can be read in one of two modes:

* Normal mode, involving translation from IBM 029 card code to ISO internal code.

* Column binary mode, in which the punchings in each column of each card are stored in 12 bits in the file.

When reading in normal mode, the file produced is a standard EMAS character file (see chapter 11). By default the following rules apply:

* All 80 columns are read.

* Translation is carried out according to the table in Appendix 2.

* Trailing spaces are deleted; that is, all the spaces following the last non-space character on a card are not included in the file.

* A NEWLINE character (ISO 10) is inserted in the file to signify the end of each card.

* The double quote character is treated as a delete character and it and the character preceding it are not transferred to the file. A series of double quotes can be used to delete preceding characters, as far back as the beginning of the card.

As explained below, some of these actions can be altered by including appropriate keywords on one of the JCL cards.

When reading in column binary mode a standard data file is produced with a fixed record length of 160 bytes (see chapter 11). Each record in the file contains 80 short integers, one for each column. The punchings in a column are represented in the least significant 12 bits of the corresponding short integer, with one bit set for each hole punched. The mapping used is:

Card Row	12	11	0	1	2	3	4	5	6	7	8	9
Corresponding bit	11	10	9	8	7	6	5	4	3	2	1	0

The bits in the short integer are numbered from right to left, starting with bit 0. Bits 12 to 15 are set to 0.

Using this mode, any combination of punchings can be read.

Paper tape input

Paper tape can be read in one of two modes:

* normal mode, for reading ISO-coded 8-track even-parity tape
* binary mode, for reading any punchings on 8-track tape

When reading in normal mode the following rules apply:

* all characters with odd parity are converted to the SUB (ISO 26) character
* the parity bit is set to 0 on input, whatever its value on the paper tape
* null (no holes punched) and delete (all holes punched) are ignored
* CARRIAGE RETURN (ISO 13) characters are ignored
* trailing spaces (SPACE characters immediately before a NEWLINE character) are ignored

As explained below some of these actions can be altered by including keywords in one of the JCL statements.

When reading in binary mode all characters, including run-out (0), are read into the file, which is created as a standard data file (chapter 11) with a fixed record length of 80 bytes.

Job control for file input

The following JCL statements are needed for file input:

```
//username   FILE   (PASS=back)
//filename   DD   *
    <contents of file>
//
```

where	username	is the name of an accredited EMAS user
	back	is the user's background password (see chapter 20)
	filename	is the name to be given to the file (see chapter 3)

Notes

* The format of the statements is fixed. Each statement should start on a new line; this means that for paper tape input there should be a line feed character before the first statement and between statements. One or more spaces should be inserted where indicated, but nowhere else.
* The name chosen for a file should not be the same as the name of an existing file. If it is, the existing file will be left and the one being read in will be ignored.
* Files are CHERISHed automatically on input.

* More than one file can be input, the form then being:

```
//username   FILE   (PASS=back)
//file1   DD   *
   <contents of file1>
//file2   DD   *
   <contents of file2>
   .
   .
   etc.
   .
//
```

* When reading in normal mode the following options can be used. The options selected should follow the asterisk on the DD card, separated from it and each other by commas:

  * NOIDENT - ignore the contents of columns 73-80; this is useful for removing sequence numbers

  * QUOTES - do not do double quote deletion; i.e. double quotes in the input stand for themselves

  * TRAIL - do not delete trailing spaces

  Example

  ```
  //TEXT DD *,NOIDENT,QUOTES
  ```

When reading cards in column binary, or paper tape in binary mode, the following form is used:

```
//username   DD   (PASS=Back)
//filename   DD   BINARY
<PATTERN>
   contents of file
<PATTERN>
//
```

On the line following the DD statement the user should give some sequence of characters which he knows does not appear in his data. The data will be read until a line containing only that pattern is detected again. Note that the two pattern lines are not transferred to the file. The options described above do not apply when reading in BINARY mode.

OUTPUT

Information can be sent to output devices by two methods:

* By an explicit Subsystem command such as LIST or SEND (see chapter 8).

* Indirectly by using the DEFINE command to link a logical output channel to a particular device. The effect of this is to put the output in a temporary file which is sent to the output queue automatically when the file is closed (see chapter 11).

In neither case is the device accessed directly by the user. His output is held in an output queue until the required device is available; it is then listed. This may take place minutes or even hours after the user has requested the action. The command QUEUES is available to tell the user whether any files of his are waiting in output queues (see chapter 21).

Output device mnemonics

Output devices are referred to by mnemonics, for example .LP for the line printer. The dot is used to distinguish the device from a file name, since in many situations a device mnemonic or a file name can be used for a particular parameter in a command. If the output device is not connected directly to EMAS then its mnemonic is followed by the number of the remote terminal to which it is connected; e.g. .LP15 is the line printer connected to terminal 15. Note that since the terminal numbers are liable to change a list is not included in this manual. It can be found in the current HELP information.

Table of output devices

The following table gives the names and mnemonics of available output devices, with file types and record formats required (where relevant). File types and record formats are described in chapter 11. The final column gives the maximum size of file that can be sent to the device, in pages (each of 4096 bytes).

Device	Mnemonic	File type	Record format required	Max file size (in pages)
Line Printer	.LP	Character or Data		255
Line Printer with upper and lower case	.LLP	Character or Data		79
Money Line Printer	.MLP	Character or Data		255
Card Punch	.CP	Character or Data		48
Binary Card Punch	.BCP	Data	F160	48
Paper Tape Punch	.PP	Character or Data		48
Binary Paper Tape Punch	.BPP	Data	F80	48
Graph Plotter	.GP	Data	F80	48
Graph Plotter for liquid ink jobs	.SGP	Data	F80	48
Matrix Plotter	.MP	Data	F300	48

Notes

* For remote devices there is a limit of 79 pages on the size of output files. It should be noted that because of limitations in the communications mechanisms binary files are expanded before being sent, so the effective limit is half of this.

* The record format is described fully in chapter 11. In this context it is the format that should be used when writing files to be sent to the specified device. Additionally it is the record format implied by default when using DEFINE for these devices. For example, if the command

```
DEFINE(SQ27,.BCP)
```

  is used then each record written on sequential file 27 must be 160 bytes long.

* The significance of '.MLP' is explained below.

CHARACTERISTICS OF INDIVIDUAL OUTPUT DEVICES

Line Printer

* If the device is defined as .LP or .MLP then lower case letters are converted to upper case, if the device does not print lower case letters.

* Lines longer than 132 characters are split and continued on the following line.

* CR (carriage return) is ignored if it is adjacent to line feed. It can be used to achieve over-printing if it appears within text. Note however that not all line printers accessible from EMAS are able to do this.

* If the device is defined as .MLP then the ISO character 33 is printed as pound sterling £, instead of hash #.

Card Punch

* If the device is defined as .CP then a translation is performed from internal code to IBM 029 card code (see Appendix 2).

* If the device is defined as .CP then any lines longer than 80 characters are split and continued on the next card.

* If the device is defined as .BCP then the information is punched as column binary output. The 160 byte record is treated as 80 short integers, of which the least significant 12 bits of each are mapped onto card rows, as shown below:

Bit number	11	10	9	8	7	6	5	4	3	2	1	0
Card Row Punched	12	11	0	1	2	3	4	5	6	7	8	9

The bits in the short integer are numbered from right to left, starting with bit 0. Bits 12 to 15 are ignored.

Paper Tape Punch

* When the punch is defined as .PP, paper tape is punched using the characters sent, made up to even parity where necessary by punching in the 8th hole. Characters with values greater than 127 are converted to the SUB character (ISO 26).

* When the punch is defined as .BPP all eight bits of each byte in the file are punched on the tape.

Graph Plotter

The graph plotter should be accessed via the graphics routines provided; these are described fully in reference 8. The device .SGP is used to indicate that 'special' facilities are required, such as liquid ink.

Matrix Plotter

The software for operating this device is currently being written. Further information should be obtained from the Advisory Service.

CHAPTER 5
INTERACTIVE TERMINALS

The primary method of accessing EMAS is via an interactive terminal. Initially all interactive terminals connected to the system were teletypes - hence the mnemonic .TT. More recently a variety of hard copy terminals and cathode ray tube display terminals have been added.

Method of connection

As explained in chapter 2, interactive terminals are connected to EMAS via small computers called Terminal Control Processors (TCP). These, in turn, are connected to a Network Control Processor (NCP), which is itself connected to both the main EMAS computers. In due course this NCP will be connected to other computers to enable users to access them from the same terminals as they use to access EMAS.

Direct connection and dial-up connection

Most terminals are connected permanently to a TCP and can be used immediately to access EMAS. Others are connected via a telephone, a Modem (Modulator/Demodulator) or acoustic coupler, and a GPO telephone circuit to the switched public network. Before this arrangement can be used to access EMAS it is necessary to establish a link to a TCP, by dialling the appropriate telephone number (031-667 1071) and switching the telephone to 'data'. The precise method of doing this varies from one terminal to another and information should be sought from the person responsible for the terminal.

Mode of communication

All terminals are connected in full duplex mode. In this mode input from the terminal and output to the terminal are quite separate and can be simultaneous. This has two effects:

* Normally the characters typed on the terminal keyboard are printed immediately on the terminal. This is done by the TCP, which reads the characters typed and then 'echoes' them back to the terminal. It is however possible for the TCP to respond with different characters on the terminal from those typed. For example, during log on this facility is used to suppress printing of the password.

* It is possible to type input whilst the terminal is printing output. The input characters are stored and echoed when the output line has been printed.

APPEARANCE OF TERMINAL

The detailed characteristics of the terminal are determined by the terminal itself and the TCP to which it is connected. Below are described the details of the facilities provided by a standard TCP.

Control character functions

The following table gives those characters which have control functions. On some terminals they have individual keys. On others they are produced by holding down CONTROL (CTRL) and then typing a letter. For example

    End Message = EM = CTRL+Y

Character	Type as CTRL+	Effect	Response	Message	Notes
CR		Terminate current line	CR-LF	sends LF	
EM	Y	Terminate current input	* CR-LF	sends EM	
CAN	X	Cancel current line	↑CR-LF		
ESC		Escape to INT:			1
HT	I	Tabs to next tab position if available Tabs set at 6,9,12,15,18, 40,45 (Thus HT at start of line causes 6 spaces)	spaces	spaces	
DEL (RUBOUT)		Cancel previous character	\char		2
SOH	A	Enter SET mode			3

Notes

1. The effect of ESC is to discard any characters typed on the current line and to prompt 'INT:'. The reply should be:

   * CR to ignore the INT:. This should be used when ESC is pressed in error.

   * A single letter followed by CR. This is interpreted by the director or the Subsystem; see chapter 6.

   * Text of between 2 and 15 characters followed by CR. This constitutes a user interrupt which can be detected by using the IMP function TESTINT (see chapter 15).

2. When DEL is pressed the terminal enters delete mode and outputs '\' followed by the previous character, which is deleted from the input line. Each successive DEL deletes an earlier character and echoes it, as far back as the beginning of the line. When a character other than DEL is struck the terminal echoes '\', exits from delete mode, and the character itself is then echoed.

   For example if the third and fourth characters of DELIVER were typed in the wrong order and then corrected, the output would look like this:

   ```
   DEIL\LI\LIVER
   ```

3. SET mode is used to send commands to the TCP itself to change characteristics of its operation. The effect of typing SOH is to discard any characters on the current line and to prompt 'SET:'. The following replies are valid; each is followed by CR.

Reply	Effect
U	Upper Mode - All lower case letters are translated on input by the TCP to upper case. (default)
L	Lower Mode - No lower case translation is carried out on the input.
G	Graphic Mode - No lower case translation is carried out, and, with respect to terminal output, all format controls are disabled, thus allowing any character [values in the range 0-255] to be sent to the terminal.
Pn	Pads - n should be a number in the range 0-9. Used to specify number of pad characters to be inserted at start of each output line. Normally 0. It is only needed for a few special terminals, which require a delay to allow the carriage to return to the beginning of the line.

Type ahead

An important characteristic of the terminal support mechanism on EMAS is the ability for a user to type ahead. This means that it is not necessary to await the completion of one operation before typing the next command. There are a number of points to bear in mind:

* When typing ahead it must be appreciated that mistakes in typing earlier commands can have disastrous results later on. It is suggested that until users are conversant with the system they await the outcome of each command before typing the next.

* Because the prompt is not printed (see below) and because output and input will be interleaved, it is not always easy to decipher a listing produced on a hard copy device, such as a teletype, when extensive type-ahead is used. The RECALL facility (see chapter 12) avoids this problem.

Prompt mechanism

Whenever input is requested by the system or by a running program a prompt is output on the interactive terminal. This acts as a reminder to the user that input is required, and can also serve to indicate what sort of input is required. The prompt text is set by

* the Subsystem

* a Subsystem facility, e.g. EDIT

* a user program calling the routine PROMPT or FPRMPT (see chapters 15 and 16)

Note that the prompt is not output if the required input has already been typed ahead.

Logging in

Before a user can log on to the system from an interactive terminal he has to obtain an accredited username (see chapter 20). The username has associated with it two passwords (see chapter 20). The first of these is used for interactive access. In order to log on the following steps should be taken:

* Switch on the terminal, and, if using a dial-up line, also the modem or acoustic coupler.

* Switch the terminal to full Duplex.

* If using a dial-up line dial the correct number (031-667 1071) and if a data tone is heard (high pitched tone) switch to data. (The exact method depends on the terminal and MODEM or coupler being used.)

* Press the space bar. If there is no response pres the CR key.
* To the prompt 'HOST:' reply EMAS.
* To the prompt 'USER:' reply with the username.
* To the prompt 'PASS:' reply with the foreground password.

All three replies should be terminated by CR.

In response the TCP and NCP will attempt to log the terminal on to the appropriate EMAS main computer. If this is successful the message

```
PROCESS STARTED
```

will be printed, sometimes followed by a message of the day, and eventually a prompt COMMAND: will appear.

Alternatively one of the following messages, or some other explanatory message, will be printed:

SYSTEM FULL	- try later.
CANNOT START PROCESS	- possibly because the process is just stopping - try again, and if the problem persists contact the Advisory Service.
PROCESS RUNNING	- a background job is running which you DETACHed earlier (see chapter 19), or another person who shares the username is currently logged on. Try later.
NO USER SERVICE	- this indicates an attempt to log on outwith the service period, or a machine fault. Ring the answering service (031-667 7491) for information.
INVALID USER	- this may be caused by mis-typing the username or because one of the EMAS computers is unavailable (see above).
INVALID PASSWORD	- this means that the password was typed incorrectly.

CHAPTER 6
INTRODUCTION TO THE SUBSYSTEM

This chapter provides an introduction to the standard EMAS Subsystem. It explains the Subsystem's command language, describes its interactive terminal interrupts, introduces the file types provided, gives sources of further information, summarises its functions in logical groups and indicates how these are covered by the rest of this manual.

The standard Subsystem

The phrase 'standard Subsystem' is used to emphasize that the Subsystem described is the one provided as a standard part of EMAS. It is not, however, the only Subsystem and it should be appreciated that it is possible to use subsystems which differ slightly or even fundamentally from the standard one, without interfering with other users, and without having to make changes to other components of the system. Chapter 18 shows how it is possible to add commands to the standard Subsystem. More fundamental changes require information outwith the scope of this manual.

The Subsystem command language

The Subsystem command language is used to communicate with the Subsystem, both from interactive terminals and from background jobs (see chapter 19). Commands are typed according to the following rules:

* Each command should start on a new line.
* If the command requires one or more parameters, these should be typed after the command, enclosed in parentheses.
* Parameters should be separated by commas.
* To indicate that a parameter has been omitted an extra comma should be inserted, if more parameters follow.
* Spaces and newlines within parameters are ignored.
* After removing spaces, newlines and parentheses the total length of the parameters should be not greater than 63 characters.

Examples of commands

```
ALERT
LIST(ABC)
LIST(ABC,.LP)
FLIST(,ALL)
DESTROY(ABC,DEF,GHI)
```

In the rest of this manual the following format is used for commands:

```
COMMANDNAME(PARAMETER1[,PARAMETER2])
```

Note that the square brackets are used to indicate optional parameters - they are not typed when the command is used.

The full details of the parameters required for each command are given with the description of the command. There are, however, a number of common features:

* List - this is used for commands that can operate on a number of items of the same type. The list can consist of one item, or more than one, in which case all but the last are followed by a comma. For example, DESTROY can be used with a list of filenames:

  ```
  DESTROY(NABC)
  DESTROY(NABC,TYPE,FILE27)
  ```

* Output device - this is used for commands that generate output. In almost every case the default output device is the interactive terminal. Output devices are described in more detail in chapter 4. The name of a device consists of a full stop followed by a mnemonic; for example, .LP means line printer:

  ```
  LIST(TEST27,.LP)
  ```

* Output file - this is often an alternative to an output device. It is sometimes useful to direct the output from a command into a file, for subsequent examination using the editor or a user program. For example, the command FILEANAL can be used to obtain information about a file. If a second parameter, not an output device, is given then the information is put in the file named by the second parameter:

  ```
  FILEANAL(FILEABN,OUT)
  ```

  Other parameters are described in the context of particular commands.

Messages output by the Subsystem

In general, simple commands do not produce any output if they work successfully. A few, indicated in the table at the end of this chapter, produce confirmatory messages. Even these may be suppressed if preferred by use of the OPTION facility - see chapter 21. All commands produce failure messages if they do not work correctly. Subsystem failure messages are described in two places:

* Messages specific to a particular command are described with that command

* General error messages are described in Appendix 1

Operator messages

Apart from messages generated by the Subsystem there are messages sent to interactive terminals by the EMAS operators, or on their behalf. These messages are of the form:

```
**OPER hh.mm message
```

It is hoped that these messages will be self explanatory, but since they are restricted to 19 characters in length they are likely to be somewhat terse.

CONSOLE INTERRUPTS

Apart from normal interactive terminal input and output there is a mechanism whereby any operation can be interrupted. The method, described in chapter 5, allows the user to input a message of up to 15 characters. Single character input messages are used to control the Subsystem in the following manner:

Interrupt	Effect
A	Abort current command or program and return to read the next command.
C	As for A except that additionally any input that has been typed ahead is lost.
H	Use during printing of diagnostics to return to calling program. This is only of use when diagnostics have been printed as a result of a call of %MONITOR from IMP or DIAG in FORTRAN.
M	Make disc consistent; i.e. ensure that all copies of files in immediate store include all changes made to date. It can be used at any time without otherwise affecting the command that is currently being obeyed.
Q	Abort current command, print diagnostics and return to command level.
T	Print out the time and number of page turns since the start of this command and the number of users logged on, without affecting command being executed.

SUBSYSTEM FILE TYPES

There are five types of file recognised by the Subsystem. The type of a file is determined from information held at the beginning of the first page of the file. The file types are listed below with an indication of the main chapter describing their use.

Type	Use	Chapter
CHARACTER	Contains characters - e.g. program source.	11
DATA	Contains binary data in discrete records.	11
OBJECT	Contains compiled programs and routines.	9
LIBRARY INDEX	Contains information to associate routine entry names with OBJECT files - used by the loader.	10
STORE MAP	Unstructured file used for direct mapping of data.	13

SUBSYSTEM INFORMATION

Apart from this manual the primary sources of information about the Subsystem are:

* the commands HELP and ALERT
* the EMAS Information Card
* the ERCC Advisory Service

The command HELP

This command provides on-line information for EMAS users. If the command is typed with no parameter then the output is a list of current commands and a brief description of their purposes. If a parameter is given which is the name of one of these commands then fuller information about the chosen command is typed. For example

```
HELP(FILEANAL)
```

would give information about the command FILEANAL.

Apart from commands, there are a number of general headings about which information is available. Currently these are:

```
ADVISORY
FILE INDEX
FILES
GRAPHICS
INPUT
INTERRUPT
LIBRARIES
MAGNETIC TAPE
NEWS
REMOTES
SCHEDULE
TERMINAL
```

Finally, HELP can be used with a parameter '.LP', '.LLPnn' or '.LPnn' (see chapter 4), in which case the whole of the current HELP text is printed on the local line printer or remote line printer specified.

The command ALERT

This command provides information about recent changes to the system and any serious faults that have been reported or corrected. If the command is typed with no parameter then the output is given on the interactive terminal. Otherwise it can be directed to a local or remote line printer; for example:

```
ALERT
ALERT(.LP)
```

EMAS Information Card

This quick reference information card provides a list of the currently available commands and their parameters and also details of remote terminals. It is intended to reprint it at least once each year, so it should reflect changes more quickly than this manual. Copies are available from the ERCC Library.

ERCC Advisory Service

The Advisory Service is available to users of EMAS, and Advisors will endeavour to answer questions about the Subsystem and the main programming languages.

Full details of the Advisory Service are contained in the current edition of the ERCC Advisory Guide.

Subsystem facilities

The facilities and commands provided by the Subsystem are divided in this manual into the following groups:

* General File Utility commands - these commands operate in respect of files as units. They operate on any type of file.
* Type Specific File Utility commands - these are used to carry out functions such as copying and listing files.
* Compilers and associated commands.
* Commands associated with program loading.
* Commands related to manipulating user data.
* File editing commands.
* Commands concerned with running work in background mode.
* Commands concerned with accounts and usage.
* Information and other commands that do not conveniently fit into other categories.

The table following gives a list of the commands in each group and for each the following information:

* A brief description of the purpose of the command.
* Whether the command produces any output (other than a failure message).
* A page number in this manual of the main description of the command.

TABLE OF COMMANDS

Group	Command	Purpose	Output	Page
General File Utilities	ACCEPT	Transfer file from another user		38
	ARCHIVE	Mark file for transfer to archive store		40
	ARCHLIST	Obtain list of files in archive store preparatory to deleting some	*	42
	CHERISH	Mark file for backing-up		40
	DESTROY	Destroy file		36
	DISCONNECT	Remove file from virtual memory		37
	FINDFILE	Locate file(s) in archive store	*	41
	FLIST	Obtain list of files in immediate store	*	35
	HAZARD	Un-CHERISH file		40
	OFFER	Mark file for transfer to another user		38
	PERMITFILE	Allow other users access to a file		38
	RENAME	Change the name of a file		36
	RESTORE	Copy a file from archive to immediate store	*	41
Type Specific File Utilities	CONCAT	Join two or more character or data files	*	46
	COPYFILE	Copy a file	*	46
	FILEANAL	Obtain details of type, contents and access permission of a file	*	45
	LIBANAL	Obtain details of contents of library index file	*	59
	LIST	List file on output device	*	47
	SEND	List file on output device and destroy it	*	48
Compilers and associated commands	ALGOL	Compile ALGOL 60 source file	*	107
	FORTE	Compile FORTRAN IV source file	*	99
	IMP	Compile IMP source file	*	89
	LINK	Join two or more OBJECT files	*	52
	PARM	Set compiler options		51
Manipulating Data	CLEAR	Break link set up by DEFINE or DEFINEMT		67
	DDLIST	Print list of current logical channel definitions		68

Group	Command	Purpose	Output	Page
Manipulating Data contd.	DEFINE	Set up link between logical channel and particular file or output device		63
	DEFINEMT	Set up link between logical channel and user magnetic tape file		86
	NEWSMFILE	Create new file to be accessed via store mapping facilities		79
File Editing	EDIT	Edit character file	*	69
	LOOK	Examine contents of character file	*	75
Program Loading and Execution	APPENDLIB	Nominate additional library index for searching during program loading		56
	INSERTFILE	Insert details of object file in current library index		55
	PERMITLIB	Allow access to a library index and inserted files		59
	REMOVEFILE	Remove reference to object file from current library index		56
	REMOVELIB	Remove library index from current search list		56
	RUN	Execute program		53
	USERLIB	Nominate new library index	*	55
Background Mode	DELETEJOB	Remove job from background job queue	*	117
	DETACH	Put job into background job queue	*	115
	FINDJOB	Find information about jobs in background job queue	*	117
Commands associated with accounting	METER	Print usage information for current session	*	121
	PASSWORD	Change foreground and/or background password		119
	PROJECT	Set project code		120
	USERS	Print number of currently active users	*	121
Information and other commands	ALERT	Obtain information on state of System	*	30
	CPULIMIT	Set time limit for each command	*	123
	DELIVER	Set text for heading of line printer output, etc.	*	124
	HELP	Get advice on using Subsystem	*	30
	OBEYFILE	Execute a sequence of commands	*	124

Group	Command	Purpose	Output	Page
Information and other commands contd.	OPTION	Set user options		124
	RECALL	Examine file containing record of interactive terminal I/O	*	76
	QUEUES	Print information about files waiting in output queues	*	125
	STOP	Terminate foreground session	*	125
	SUGGESTION	Send suggestion to System Manager		126

CHAPTER 7
GENERAL FILE UTILITY COMMANDS

In chapter 3 the concept of a file was introduced and the conventions relating to EMAS files were described.

As explained there, the basic file handling facilities are provided by the director. These facilities act on files as sequences of pages, the contents of which are not significant. Thus, for example, to the director there is no distinction between a file containing character data and a compiled object file. This chapter describes the file manipulation commands provided by the Subsystem which make calls on the director, and which act on all types of file. The different types of file provided by the EMAS Subsystem are described in the chapters following.

The command FLIST

FLIST is used to find the names of all the files belonging to this user. Optionally, extra information can be obtained about each file and some summary information about all files. FLIST provides no information about the contents of a file - the command FILEANAL (chapter 8) should be used instead.

The form of the command is

```
       FLIST([out][,options])
```

If the command is typed without a parameter then the filenames of all user-created files belonging to this user are printed on the interactive terminal, in alphabetical order one to a line. Those that are CHERISHed (see later in this chapter) are preceded by an asterisk. For example

```
       * ABC
       * CTEST
         OBJC
         TEMP
```

Instead of output being set to the interactive terminal it can be directed to an output device or to a file, by typing the device or filename as the first parameter; e.g.

```
       FLIST(.LP)
       FLIST(FLFILE)
```

Note that if a file of the given name already exists, the command will fail; i.e. a new file should always be specified.

The second parameter can be 'V' if it is required to print the filenames across the page. This is particularly useful when using a video terminal which displays a limited number of lines. Alternatively the second parameter can be 'ALL', in which case the output takes the following form:

```
USER: ERCC06
  4 USER FILES + 4 SUBSYSTEM FILES
  76 PAGES (9 CHERISHED)
FILE LIMITS: 64
FREE LIST CELLS: 95

    FILE            PAGES         OWNP          EEP           ACC
*   ABC                8           15            0             0
*   CTEST              1           15            8             0
    OBJ                1           15            0             0
    SS#BGLA           16           15            0             3
    SS#GLA            16           15            0             3
    SS#LIB             1           15            0             2
    SS#STK            32           15            0             3
    TEMP               1           15            0             2
```

The purpose of the list cells is described in chapter 3. The headings have the following meanings:

PAGES	size of file in pages.
OWNP	access permission for owner. This is a four bit field where the bits take the following meanings:

2^0	WRITE
2^1	READ
2^2	WRITE SHARED
2^3	READ SHARED

In the example above all files are permitted to their owner in all modes.

EEP everyone else's access permission. The bits have the same meaning as in OWNP. In the example above the file CTEST is permitted to everyone else in READ SHARED mode (see PERMITFILE below).

ACC this indicates the current connect mode of the file. If it is not connected in the owner's virtual memory it has a value of 0. Otherwise its value is constructed from the following bits:

2^0	WRITE
2^1	READ
2^2	SHARED

Hence, in the example above, the file TEMP is connected in READ un-shared mode.

Further information about access permissions and connect modes is given in chapter 3, and later in this chapter.

CREATING, RENAMING AND DESTROYING FILES

Files can be created, renamed or destroyed only by their owners. There is no general command used to create a file. Files are created as a result of using certain commands or facilities. For example the command EDIT can be used to create a new character file. If this file is compiled using the command IMP an object file may be created. If the output from FILEANAL is directed to a file and a file of that name does not exist, one will be created. In general the following rule applies in relation to commands that create files:

> If a file of the requested name exists already, it is overwritten - destroying any information it currently contains. If not, then a new file is created with the requested name.

The command RENAME

A file can be renamed using the command RENAME. This takes two parameters:

```
RENAME(oldname,newname)
```

oldname is the current name of the file
newname is the name to be given to the file

RENAME will fail if a file with the name 'newname' already exists, or if the file being renamed does not exist or is connected in another user's virtual memory or is on OFFER (see below). Note that access permissions and the cherish status of the file are not affected by renaming.

The command DESTROY

One or more files can be destroyed by a call of DESTROY. It takes the name of one or more files as its parameter(s):

```
DESTROY(ABC)
DESTROY(TEMP,COBJ,BACL3)
```

The command will fail if the file being destroyed is connected in another user's virtual memory or is on OFFER (see below). Also if a file is permitted to its owner with an access permission of 0 (i.e. no access at all) it cannot be destroyed. This fact could be used to protect a file from inadvertent destruction - but see also CHERISH below.

CONNECTING AND DISCONNECTING A FILE

Before any use can be made of the contents of a file, e.g. before a character file can be edited or an object file can be executed, it must be connected in the user's virtual memory. This operation is described in chapter 3. There is no general command for this purpose - connection occurs as a result of the use of a wide variety of commands or facilities. For example, a file is connected when:

* it is analysed by FILEANAL
* it is listed on the interactive terminal using LIST
* it is edited
* it is read from by a FORTRAN program

Normally, once a file has been connected, it remains connected for the rest of the session - i.e. until the user logs off. There are however a number of commands which cause disconnection, and there is also an explicit DISCONNECT command. The following commands disconnect the file on which they are operating if it is connected at the time:

```
DESTROY
RENAME
OFFER
PERMITFILE
PERMITLIB
SEND
COPYFILE (disconnects the input file if it belongs to another user)
```

The command DISCONNECT

This command can be used to disconnect one or more files from the user's virtual memory. It takes as its parameter the name of one or more files that are currently connected. There are several situations in which it is useful:

* To protect the file. By disconnecting it the user can be sure that its copy on immediate store (the discfile) is up to date (this can also be achieved by INT:M - see chapter 6). Also, since it is no longer connected in the virtual memory it cannot be corrupted by programs being run by this user that might be faulty. In fact this form of corruption is unlikely, since user files are normally left connected in READ mode (and are therefore protected) when they are not currently being used for output.
* To free the file for use by another user. For example, after a user has executed an object file belonging to another user, the file will remain connected in his virtual memory. If the owner attempts to alter the file (by re-compiling it) a failure will occur, because it is not possible to write to a file connected in READ mode in another virtual memory. If the user who has run the program DISCONNECTs the file, the recompilation will then be possible.
* To free space in the virtual memory. Although large (13 Mbytes) the virtual memory can be filled during a session. To free some space the user should use DISCONNECT to disconnect some of the files that are no longer being used.

TRANSFERRING OWNERSHIP OF A FILE

The two commands OFFER and ACCEPT can be used to transfer a file from one user to another. The owner of the file should use the command OFFER, which takes two parameters: the name of the file to be transferred, and the name of the user to whom it is to be transferred.

```
        OFFER(ABC,ERCC98)
```

would offer the file 'ABC' to user 'ERCC98'.

Note that once a file is on it cannot be connected in any virtual memory, regardless of access permissions

An OFFER can be revoked, if necessary, by using the command OFFER with only one parameter - the name of the file.

A file can be offered to any user on either machine.

Accepting the file

The user to whom the file is offered can accept it at any time by using the command ACCEPT. This takes as its first parameter the full file name of the file to be accepted. For example, if the user OFFERing the file in the example above was ERCC38 then the user ERCC98 would type

```
        ACCEPT(ERCC38.ABC)
```

The effect of this would be to transfer the file 'ABC' from user ERCC38 to user ERCC98, giving it the new name ERCC98.ABC. This command will fail if user ERCC98 already has a file 'ABC'. However this problem can be overcome by typing a second, optional, parameter to ACCEPT, which is the new name to be given to the file. For example,

```
        ACCEPT(ERCC38.ABC,NEWABC)
```

In this case the file will be transferred and given the new name ERCC98.NEWABC.

SETTING ACCESS PERMISSIONS ON FILES

In chapter 3 there is a description of the access permission mechanism. This section describes the use of the command PERMITFILE. This command, which can only be used in respect of one's own files, takes three parameters:

```
        PERMITFILE(file,user,mode)
```

file is the name of a file belonging to this user

user is one of the following

null	meaning give access to all other users
a username	meaning give access to a particular user (can be the owner)
a user-group	meaning give access to a group of users. The given parameter should contain up to 5 '?' characters. For example, EGNP?? means give access to any user with a username containing 'EGNP' as its first four characters.

Mode is one of the following modes:

RS or null	READ SHARED
R	READ
WS	WRITE SHARED
W	WRITE
NONE	no access.
CANCEL	used to cancel an access permission given previously to an individual user (other than the owner) or a user group.
ALL	all modes (see below).

In some situations it is necessary to combine more than one mode. This is done by using a single hexadecimal digit for the mode. This consists of a four-bit field, where the bits have the following meanings

2^0	WRITE
2^1	READ
2^2	WRITE SHARED
2^3	READ SHARED

The table below shows the complete range of possible combinations:

MODE	UNSHARED		SHARED	
	WRITE	READ	WRITE	READ
0 (NONE)				
1 (W)	*			
2 (R)		*		
3	*	*		
4 (WS)			*	
5	*		*	
6		*	*	
7	*	*	*	
8 (RS)				*
9	*			*
A		*		*
B	*	*		*
C			*	*
D	*		*	*
E		*	*	*
F (ALL)	*	*	*	*

Notes

* When a file is created it has default access permissions of all modes to its owner and no access to anyone else.

* There is no overhead associated with access permissions to self and everyone else. Permissions to individuals and groups, however, require space in the file index (see chapter 3).

Examples

```
PERMITFILE(ABC)
```

This permits the file to everyone else with the default access permission READ SHARED.

```
PERMITFILE(DOUBLE,ERCC23,WS)
```

This permits the file DOUBLE to user ERCC23 with WRITE SHARED access permission.

Multiple permissions

It is possible to use PERMITFILE more than once in respect of a file. For example in the following sequence a file is permitted to all users with READ SHARED access permission, but access is withdrawn from users with user numbers starting with 'Y'. Finally access in all modes is granted to ERCC28.

```
        PERMITFILE(PERTEST)
        PERMITFILE(PERTEST,Y?????,NONE)
        PERMITFILE(PERTEST,ERCC28,ALL)
```

Note that even if a file is permitted to another user in WRITE mode he cannot alter its size.

The command PERMITLIB (see chapter 10) can be used to control the access permissions of library index files and the object files to which they refer.

COMMANDS RELATED TO BACKUP

As explained in chapter 3 some files are copied onto a back-up store. All files which are likely to be difficult to reconstruct in the event of file system corruption should be marked by use of the CHERISH command. This command can be used to mark one or more files:

```
        CHERISH(SNAP)
        CHERISH(ABC,MINE,COBJECT)
```

The command HAZARD can be used to remove the CHERISH status.

Notes

* When first created, files are not normally CHERISHed. It is the user's responsibility to CHERISH his important files.

* The CHERISH status of a file also affects its disposal when it is left unused for a significant period - see below.

COMMANDS RELATED TO ARCHIVE STORAGE

The archive store is held on magnetic tape, quite separately from the backup store. It contains files that have been moved there from immediate storage for one of the following reasons:

* Because the owner has indicated that he wishes the file to be moved by using the ARCHIVE command.

* Because the file has not been used for a significant period (currently about four weeks) and it has been moved by the system in order to free space in the immediate store. Note that this only applies to files that are CHERISHed: un-CHERISHed files are destroyed if they remain unused for a significant period (currently about four weeks).

Once in the archive store there is no distinction between files moved in for different reasons.

The command ARCHIVE

This command, which takes one or more filenames as a parameter, is used to mark files which the user wants to move from the immediate store to the archive store. Note that this command does not take effect immediately: there will be a delay of up to a week before the file is moved. There are a number of reasons for using this command:

* to clear space in the file index
* to dispose of files that are not currently required but may be needed at some later date
* to reduce the charge for keeping files on the system (see chapter 20)

The command FINDFILE

This command is used to obtain information about the user's files in the archive store. It can be used in one of two modes:

* to locate all files of a particular name
* to list the names of all files belonging to the user in the archive store

In order to locate a particular file the command is typed thus:

```
FINDFILE(filename)
```

For example:

```
FINDFILE(KERN27S)
```

If a file of the required name is found, information about it will be output; e.g.

```
ERCC06.KERN27S   27/10/75   17
```

The date indicates when the file was moved from immediate store to the archive store. The last number on the line gives the size of the file in pages (4096 bytes). Since there may be more than one file in the archive store of the same name, the opportunity is given to continue scanning the directory to locate earlier copies. The reply to the prompt 'CONTINUE SEEK?' should be 'Y' or 'YES' if the user is interested in earlier copies of the file. Otherwise the reply should be 'N' or 'NO'. This sequence of printing out information about one file of the specified name and prompting for further searching continues until no further copies of the file are found when the message 'SEARCH ENDS' is typed. For example

```
COMMAND: FINDFILE(PRINT01S)
ERCC06.PRINT01S   20/10/75   12
CONTINUE SEEK?  Y
ERCC06.PRINT01S   13/04/75   11
CONTINUE SEEK?  Y
SEARCH ENDS
```

If a list of all of a user's files in the archive store is required than the first parameter to FINDFILE should be omitted. In this case an optional second parameter can be used to specify an output device or file. For example

```
FINDFILE(,.LP)
```

File information is listed in reverse chronological order, i.e. the most recent files first.

Moving files from archive store to immediate store

The command RESTORE is used to copy a file from archive store to immediate store. Note that the copy in the archive store is not altered by this command. The command takes two parameters. The first is the name of the file being restored, and the second, which is optional, is the date of archiving. This should be typed exactly as it appears in the FINDFILE output. By default the most recent copy of a file is restored. The date is only needed when an earlier copy is required.

Example

```
      RESTORE(KERN27S)
      RESTORE(IMP907A,23/12/75)
      RESTORE(DATA27,01/07/74)
```

RESTORE will fail immediately if:

- * A file of the same name already exists in the user's immediate store file index. To avoid this it is necessary to rename the existing copy before restoring the old one.
- * The date is typed in incorrect format.
- * There is no file in the archive store of the requested name, or, if a date is used, no file of the requested name is held for that date.

If the RESTORE command is successfully interpreted then a request is sent to the VOLUMES process to carry out the operation. The user can then proceed to give other commands to EMAS. The file should be recovered from the archive store within 15 minutes and if the user is still logged on an operator message will be typed on his interactive terminal telling him that the file has been restored. If he is not logged on he can check for the existence of the file with FILEANAL or FLIST when he next logs on.

The RESTORE operation can fail when the VOLUMES process attempts to copy the file to the immediate store. This will occur if:

- * There is insufficient room in the user's file index.
- * There is a file of the same name in the user's file index. This would only occur if the user had created a file of the same name after typing the RESTORE command.

If the user is still logged on an appropriate operator message will be typed on his interactive terminal.

Files restored in this way are un-CHERISHed and have default access permissions.

Destroying files in the Archive store

The present method of destroying files in the archive store involves the use of the command ARCHLIST. This command, which takes no parameter, creates a file called SS#DARCH which contains a list, similar to that produced by FINDFILE, of the names of the user's files in the archive store. The most important difference in the list is that each line begins with an asterisk. In order to delete files from the archive store the user should, by using the EDIT command or some other means, remove the asterisks from the lines specifying the files he wishes to destroy. No other alterations should be made to the file.

Within 24 hours the file SS#DARCH will be removed from the user's file index and the requested changes will be made to the directory of archived files.

For example, if after typing ARCHLIST the file SS#DARCH contained:

```
      *ERCC06.ABC        27/03/76   4   10848
      *ERCC06.NEWCA      21/03/76   4   10820
      *ERCC06.SPC03PN    21/03/76   4   10812
      .
      .
      .
```

and if the user wished to delete the files ABC and SPC03PN in the archive store he should edit the file to become:

```
      ERCC06.ABC        27/03/76   4   10848
      *ERCC06.NEWCA      21/03/76   4   10820
      ERCC06.SPC03PN    21/03/76   4   10812
      .
      .
```

Note that if any other alterations are made to the file no files will be deleted. If the command ARCHLIST is used again before SS#DARCH has been removed from the user's file index then it will fail with the message 'SS#DARCH ALREADY EXISTS'. If the user wishes to delete more files then he need only edit SS#DARCH again, removing more asterisks from the file as necessary.

It should be appreciated that FINDFILE and RESTORE work from the archive store directory. This is not altered directly as a result of editing SS#DARCH. It is only altered when this file is removed from the user's file index and processed by the system. Hence, if a user types FINDFILE immediately after editing SS#DARCH, he will find that the changes he has made will not have taken effect. Normally they will be made by the following day.

Whilst the principle of the archive and back-up stores will remain the same, their precise operation and the user commands provided for manipulating them are likely to alter in the near future.

CHAPTER 8
TYPE-SPECIFIC FILE UTILITY COMMANDS

Chapter 7 describes the Subsystem functions that operate on files as units, without regard to their contents. This chapter describes a number of type specific file utility commands that carry out simple operations on files or provide information about their contents.

The table below shows the available functions, relevant commands and allowed file types:

Function	Commands	Valid File Types				
		Character	Data	Object	Library Index	Store Map
Provide information about contents	FILEANAL	*	*	*	*	*
	LIBANAL				*	
Copy	COPYFILE	*	*	*	*	*
Join together	CONCAT	*	*			
	LINK			*		
List on output device	LIST	*	*			
	SEND	*	*			

COMMANDS FOR OBTAINING INFORMATION ABOUT THE CONTENTS OF FILES

The command FILEANAL

This command is used to obtain information about a particular file. The file may belong to this user or, if permitted to him, to any other user. The first parameter is the name of the file, and the second parameter is an output file or output device. By default output goes to the interactive terminal.

Examples:

```
FILEANAL(TEMP)
FILEANAL(ERCC27.BASEFILE,.LP)
```

For all file types the output includes the size of the file, in pages, the number of other virtual memories in which it is currently connected, i.e. the number of other users currently accessing it (if any), and all the current access permissions. The remaining information depends on the type of the file. The type is determined from the header (the first part of the first page of the file), and the following table summarises the information given for each:

Type	Information	Chapter
Character	Length (in bytes) of user data	11
Data	Length (in bytes) of user data, maximum allowed length, record format	11
Object	Routine and Data entries	9
Library Index	Files inserted and other library indexes appended	10
Store Map	Length in bytes	13
Non-Standard	None	

Note that the command LIBANAL can be used to give fuller information about library index files (see chapter 10).

COMMANDS FOR JOINING AND COPYING FILES

The command COPYFILE

This command is used to make a copy of a file. It can be used to copy any type of file, and, subject to suitable access permission having been granted, can be used to make a copy of a file belonging to another user. It takes two obligatory parameters: the name of the file to be copied, and the name of the file into which it is to be copied. For example:

```
COPYFILE(KERN27,BACKUP)
COPYFILE(ERCC27.TEST23,TEST23)
```

Notes

* If a file of the name of the new copy already exists its contents will be overwritten, but its cherish status and access permissions will be preserved. Otherwise a new file will be created, with default cherish status and access permissions.

* COPYFILE should not be used to copy a library index file from one user to another. Instead a new library index file should be created using the USERLIB command and, if required, the appropriate calls of INSERTFILE and APPENDLIB. This restriction is made because a library index file contains within itself the name of its owner.

* If successful, COPYFILE produces a confirmatory message.

The command CONCAT

This command is used to concatenate a number of character and data files and create an output character file. It reads the names of the files involved either from the interactive terminal or from a control file; in the latter case the control file name is given as a parameter to CONCAT. Hence either:

```
    CONCAT          when reading file names from the terminal
or
    CONCAT (CFILE)  when reading file names from file CFILE
```

In either case the filenames should be typed one to a line, the list being terminated with the keyword '.END'. This should be followed by a line containing the name of the output file. When reading file names from the interactive terminal the prompt is 'CONC:'. The prompt for the output file is 'FILE:'. For example:

```
COMMAND:CONCAT
CONC:ERCC27.LIST
CONC:MINE
CONC:LISTSTOP
CONC:.END
FILE:NEWLIST
```

Notes

* The files used for input are not altered by this command.
* The rules concerning the creation of output files are as for COPYFILE.
* The output file cannot have the same name as that of any of the input files.
* The output file is always a character file - regardless of type of input file.
* If any of the input files is a data file with a record format of VA or FA, the carriage control characters in it will be replaced by newline characters.
* If successful, CONCAT produces a confirmatory message.
* There is a concatenation operator '+' which can be used when specifying parameters for the commands DEFINE, DETACH and with the commands which call the compilers: ALGOL, IMP and FORTRAN.

The command LINK

This command is used in a similar way to CONCAT, but to link OBJECT files together. It is described fully in chapter 9.

LISTING FILES ON OUTPUT DEVICES

The commands LIST and SEND are used to produce listings of files on output devices, such as the interactive terminal or the line printer.

The command LIST

This command takes an obligatory parameter and three optional ones:

```
LIST(filename[,device,][,copies][,special forms])
```

filename	This should be the name of the file being listed; it can belong to this user or, if suitable access permission has been granted, to another user.
device	This should be the abbreviated name for the device - see the table in chapter 4. The default is .TT (the interactive terminal). Both local devices and those connected to the Regional Communications Network can be used.
copies	This parameter can be used to specify the number of copies to be listed. It should be an integer in the range 1-15. The default is 1.
special forms	This parameter can be used when listing files on line printers connected to the Regional Communications Network which support special forms printing. Details of the special forms codes currently available can be obtained from the Advisory Service.

Examples

```
LIST(ALIST,.LP)
LIST(MND1136)
LIST(ERCC06.FILELIST,.LP15,2)
LIST(MYPROG,.GP)
```

The LIST command does not access devices directly, other than the user's terminal. Instead it makes a copy of the file and sends it to the DEMONS process to be listed when the required output device is available. This may be minutes or hours after issuing the command. The command QUEUES (described in chapter 21) can be used to determine whether any files are still awaiting listing in the EMAS output queues.

The command SEND

The command SEND is similar to LIST but is more efficient, since it does not make a copy of the file being listed. Instead, it sends the file itself to the DEMONS process. Thus, the file is effectively destroyed by this command. When a file is not required other than to produce a listing this command should be used.

Notes

* SEND uses .LP (line printer) as its default output device; hence

  ```
  SEND(SS#LIST)
  ```

  means list the compiler default listing file on the line printer and destroy it.

* SEND cannot be used for listing a file on the interactive terminal (.TT).

* Otherwise the parameters are identical to those of LIST.

CHAPTER 9
COMPILERS AND OBJECT FILES

The standard EMAS Subsystem includes compilers for the programming languages IMP, FORTRAN and ALGOL. The relevant language manuals (references 1, 2 and 3) contain full details of the languages; the environments provided by the Subsystem for programs written in these languages are described in chapters 15, 16 and 17 respectively. This chapter introduces the commands that invoke the compilers, and the associated commands PARM and LINK.

OBJECT FILES

The compilers generate object files of a standard format. This makes it possible, subject to restrictions imposed by the parameter-passing characteristics of each language, to mix object files generated from different source languages. Hence, for example, it is possible to call an IMP routine from a FORTRAN program. This facility is described in reference 9.

Sharing code

EMAS compilers generate code which can be shared. Before this sharing can be exploited the object file has to be given READ SHARED access permission to other users. In the case of most user programs this attribute is not very important. In the case of facilities such as the Subsystem base file, heavily used packages and the compilers, however, this shareability is very important in reducing the number of pages that have to be moved between the backing store and the main store. All the users sharing an object file execute the code in that file. The same file is connected in all their virtual memories and thus the number of times any one of them is delayed waiting for a page is reduced. Clearly this sharing can only be used for code and constants. Each user must have his own area for variables and arrays.

The director code too is shared by all processes, although, as indicated in chapter 2, each process has its own director.

The fact that code is shareable imposes no restraints on the user. It is the responsibility of the compiler writer to determine which parts of a program to compile into the shared area and which to put into the unshared area.

The general linkage area (GLA)

Each user has a file, created by the Subsystem, with the name SS#GLA. This contains, as its name suggests, linkage information for each object file currently loaded; for example, the address of each routine. Since the object file may be connected at different addresses in different users' virtual memories it is not possible to share this information - hence each user must have his own copy. Apart from linkage information, the GLA also contains variables which exist all the time that the program is loaded. In IMP this includes, for example, %OWN variables, and in FORTRAN, COMMON blocks. Temporary local variables, on the other hand, are stored on the STACK (see below).

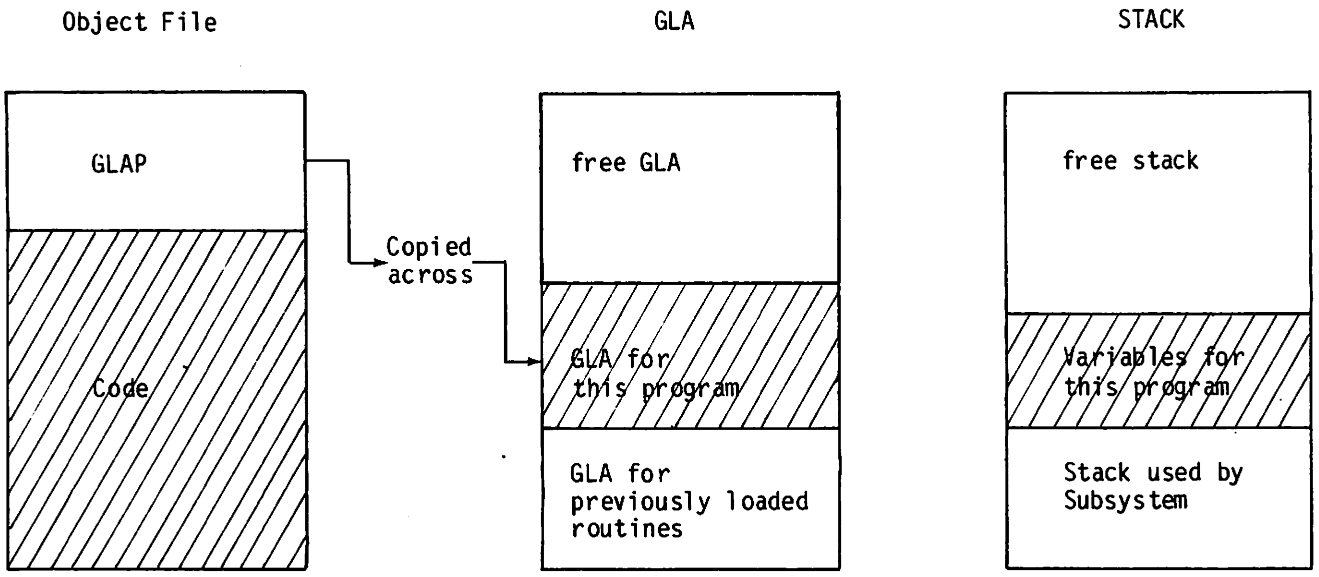
Part of the loading process (see chapter 10) involves initialising the area of GLA which is to hold the linkage information and initialised variables for the object file being loaded. This is done by copying part of the object file (the general linkage area pattern - GLAP) into the next free area of the GLA. The GLA therefore holds the accumulated linkage information and initialised variables for all the object files loaded thus far.

The stack

Each user has a file, created by the Subsystem, with the name SS#STK which contains the stack for running programs. This is the area used by IMP and ALGOL to store all local variables and arrays and by all three languages to store registers and parameters when making routine calls. The use of the stack by IMP programs is explained more fully in the IMP Language Manual - reference 4.

Summary

The three areas used at run time are as shown shaded in the diagram below.



Notes

* The object file contains primarily the code and GLAP. The file may be shared with other users. The GLAP is only used at load time to initialise the GLA.

* A separate GLA and STACK are needed by each user using the program.

Using the compilers

Before using a compiler it is necessary to prepare a character file containing the program to be compiled. This will have been read in from cards or paper tape (chapter 4) or created using the EDIT command (chapter 12). This file, known as the source file, is compiled into an object file; normally a listing file of the program is also produced. This listing contains a list of the program statements being compiled, with line numbers added by the compiler, information about any compile time faults and optionally, in the case of FORTRAN, a cross reference table for all the names used in each subroutine. By default this listing is created in a file called SS#LIST.

Prior to compilation, information about the object file is removed from the current library index, if any exists: see REMOVEFILE (chapter 10). If compilation is successful the information about the object file is put into the current library index, by an automatic call of the INSERTFILE command (see chapter 10).

The commands IMP, FORTE and ALGOL

These commands are used to compile IMP, FORTRAN and ALGOL programs, respectively.

In each case the parameters are:

1. The name of the source file containing the program or routines to be compiled. Several files can be concatenated with the + operator; for example A+B+C.

2. The name of the object file; if a file of this name exists it will be overwritten, if not a file will be created. The keyword .NULL is a valid alternative if no object file is required. This is a useful facility if the program is known to contain faults, since it reduces the compile time.

3. The name of a listing file or device. If this parameter is omitted a file created by the Subsystem with the name SS#LIST will be used. Valid alternatives include output devices (see chapter 4). Again, .NULL can be used if no listing is required. Note that SS#LIST is destroyed when the user logs off.

4. A supplementary output device, for error messages only, is available for IMP and ALGOL. This is normally used to provide a list of error messages on the interactive terminal (device code .TT). If omitted no separate list of faults is produced.

Examples of calling compilers

```
FORTE(ENGP77.FORTP4,P4Y,.LP)
```

This would compile the FORTRAN source file ENGP77.FORTP4 into an object file P4Y and would print a listing on the local line printer.

```
ALGOL(A,AY)
```

This would compile the source file A into an object file AY and generate a listing file with the name SS#LIST. Note that if SS#LIST contains a listing from a previous compilation in the current session it will be overwritten.

```
IMP(ERCC77.BASE+MINE,ABCOBJ,.LP14,.TT)
```

This would compile the source file contained in files ERCC77.BASE and MINE into the object file ABCOBJ, producing a listing on the remote line printer .LP14 and a list of compile time faults on the interactive terminal.

The command PARM

Various compile time options can be used and these are set by a call of the command PARM. A call of PARM takes effect for all future compilations until the next call of PARM or the user logs off, whichever is sooner. PARM has the effect of resetting all values to the default settings, and then setting the ones selected. Hence PARM with no parameters merely resets the defaults. The following parms are described in the language manuals:

LABELS	NOCHECK	NOTRACE
MAP	NODIAG	OPT
NOARRAY	NOLIST	STACK

Additionally the following parms are specific to EMAS:

* NOENTRIES - when this is selected information about the object file is not inserted in the current library index (see chapter 10).

* DYNAMIC - this controls dynamic loading for FORTRAN programs (see chapter 16).

The command LINK

This command can be used to link object files together to produce a compound object file. It is useful in the following situations:

* to reduce the number of separate files in the user's file index
* to ensure that a particular version of a routine is used to satisfy a particular call

The user should note however that, since program linking is automatic when a program is being run, it is usually unnecessary to use this command.

The command operates in a similar way to CONCAT (see chapter 8). It prompts for input files using the prompt 'LINK:'. The list should be terminated with '.END', whereupon the prompt 'OBJECT:' will be typed, to which the reply should be the name to be given to the single output file. The following example shows how the command can be used:

```
COMMAND:LINK
LINK:P4Y
LINK:ABCOBJ
LINK:.END
OBJECT:LINKFIL2
```

A confirmatory message is normally printed. If PARM(MAP) is set at the time of the call of LINK then a short link map is printed as well. This contains information about the relative start addresses of the object files in the combined file.

Notes

* It is not possible to extract an object file from a linked file.
* LINK can read its control information from a file, as for CONCAT (see chapter 8). The filename would be specified as a parameter to the command.

CHAPTER 10
PROGRAM LOADING AND LIBRARY STRUCTURE

Having successfully compiled a program into an object file, the next stage is to load this object file prior to executing the code contained in it. This chapter explains the mechanism by which the Subsystem loader carries out this task in respect of a complete program. As explained in chapter 18, it is possible to execute a suitably written routine as a command, and the loading mechanism described here is equally applicable to loading such a routine.

The loader is called automatically by:

* the command RUN
* a call of any command except those in the list of standard commands (see chapter 6), unless the FULLSEARCH option has been selected (see below)

The command RUN is used to load and execute a compiled program. It takes one parameter, the name of an object file which contains the compiled program.

Example:

```
RUN(MYPROG)
RUN(EGNP99.SUMXOB)
```

Before running a program it may be necessary to use the command DEFINE to establish links between I/O channels and particular files or devices (see chapter 11). Also a call of CPULIMIT may be required to set an appropriate time limit (see chapter 21).

When running a program belonging to another user, the object file must be permitted to this user in READ or READ SHARED mode. The RUN command loads the object file and then, if loading is successful, executes the code contained in the object file. The loading process involves the following sequence:

1. loading the object file containing the required program or routine
2. determining what external routines it calls
3. locating each routine and following steps 1, 2 and 3 for the object file in which it is found

This process continues recursively until all external references are satisfied. Step 1 involves the following:

* Connecting the object file in the user's virtual memory, in READ or READ SHARED mode
* Copying the initialised variables and external reference information from the general linkage area pattern (GLAP) into the next available space in the general linkage area (GLA) (see chapter 9)

Step 2 involves examining tables in the object file to determine the names of the external references it requires.

The order of searching used for step 3 is as follows:

* the current object file - because the routine might be in the same file as the routine referencing it
* the library index structure - see below

THE LIBRARY INDEX STRUCTURE

The library index structure consists of one or more library index files which contain information about object files. The existence of library indexes makes it possible to locate compiled routines automatically without the necessity for the user to nominate explicitly the object files to be used.

If, for example, a user compiled a routine R into an object file called OBJ1 and then ran a program which needed to call the routine R, it would be necessary in some way to identify the appropriate object file. There are three possible methods:

* The user could nominate, in the RUN command, the names of all the object files to be used. This would involve a lot of extra work since, for example, many programs require ten or more separate object files.

* The loader could search all object files in the user's file index. This would involve a large overhead and would be impracticable if the user wanted to use object files belonging to others. It would also prevent a user having more than one compiled version of the same routine.

* The loader could search for the routine name in one or more library index files linked in a predefined structure; having located the name a pointer associated with it in the library index file could indicate which object file to use. This is the mechanism used.

Library indexes

A library index file has all the attributes of a normal file - it can be connected in the virtual memory, renamed, destroyed, cherished and so on. It contains two sets of information:

* a table of entry points, each with a link to an object file name

* a list of names of other library index files to be searched if the entry is not found in this one

The layout of a library index file is effectively:

Entry names	Object file names
A1 RT2 TESTR	ERCC06.OBJ2
LOG ALOG	ERCC08.TESTLOG
Appended Library Index Names	
ERCC06.TESTLIB MANAGR.SYSLIB	

Notes:

* The object files and appended library indexes can belong to the owner of the library index file or to someone else.

* In order to locate an entry name quickly the entries are inserted and searched for using a hashing technique.

Size of library index

A library index file is always 1 page (4096 bytes) long. The table of entry names contains 238 cells. Each cell can contain 1 entry name or 1 file name, except that if the file belongs to another user 2 cells are required. Hence, for example, a library index might contain information about 18 different object files which contain between them 220 different entries.

Additionally, a library index can contain links to up to 16 other library indexes.

The current library index

Whilst a process is running there is always a current library index. By default this is a library index created by the Subsystem called SS#LIB. The command USERLIB can be used to select another library index.

Example:

```
USERLIB(TESTLIB)
```

If no file called TESTLIB exists then a new library index TESTLIB will be created. If TESTLIB does exist then a check will be made to ensure that it is a library index file. TESTLIB now becomes the current library index and remains so until the next call of USERLIB. Note that if USERLIB is called with no parameter it has the effect of selecting the default library index SS#LIB to be the current library index.

The current library index is used in the following ways:

* it is the first library index searched for routine entry names during the loading process

* it is the library index modified by calls of the commands INSERTFILE, REMOVEFILE, APPENDLIB and REMOVELIB

The command INSERTFILE

This command is used to insert information about an object file into the current library index. The parameter to INSERTFILE should be one or more object file names about which information is to be inserted into the library index.

Example:

```
INSERTFILE(ABC1)
INSERTFILE(FILE27,ERCC18.SSTT,MOBJ)
```

Note that although the command INSERTFILE can be used explicitly, as above, it is most commonly called (automatically) at the end of a successful compilation (see chapter 10).

Apart from general failures concerned with connecting files - see Appendix 1 - failures will occur if:

* the file being inserted is not an object file

* the file being inserted is currently inserted (use of REMOVEFILE will enable INSERTFILE to be used; this would only be necessary if the names of entries in the object file had altered since it was last inserted)

* there is a conflict between one or more entry names in the file being inserted and entry names already in the library index

* the table of entries is full

Note that a reference to a file will remain in a library index even though the file has been archived; if it is DESTROYed the current library index is modified automatically.

The command REMOVEFILE

This command is used to remove information about an object file from the current library index. The parameters should be one or more object file names currently inserted in the library index.

Example:

```
REMOVEFILE(FILE29Y)
REMOVEFILE(TEST1,TEST2,ERCC22.ABOBJ)
```

A failure will occur only if an attempt is made to remove information about a file which is not currently inserted. No check is made on the existence or type of the object files referenced.

This command is called automatically at the start of every compilation in order to remove information about the object file which is being compiled into. In this case no error message is printed if the information is not in the library index.

The command APPENDLIB

This command is used to add the names of one or more library indexes to the end of the list of library indexes in the current library index. This list determines the order of searching to be used if an entry name is not found in the current library index (see below). The command takes the library index names as a parameter.

Example:

```
APPENDLIB(TESTLIB)
APPENDLIB(MANAGR.SYSLIB,CONLIB.GENERAL)
```

This command will fail if:

* 16 library indexes have already been appended to the current library index

* the file being appended is not a library index, or is not permitted in READ or READ SHARED mode to this user

* the file being appended is already appended to the current library index

* an attempt is made to append the current library to itself

Note that no check is made to ensure against a closed loop of appended library indexes; e.g. A appended to B appended to A. This situation can cause failure during loading (see below).

The command REMOVELIB

This command is used to remove the name of one or more library indexes from the list in the current library index. The parameter should be the names of the library indexes.

Examples:

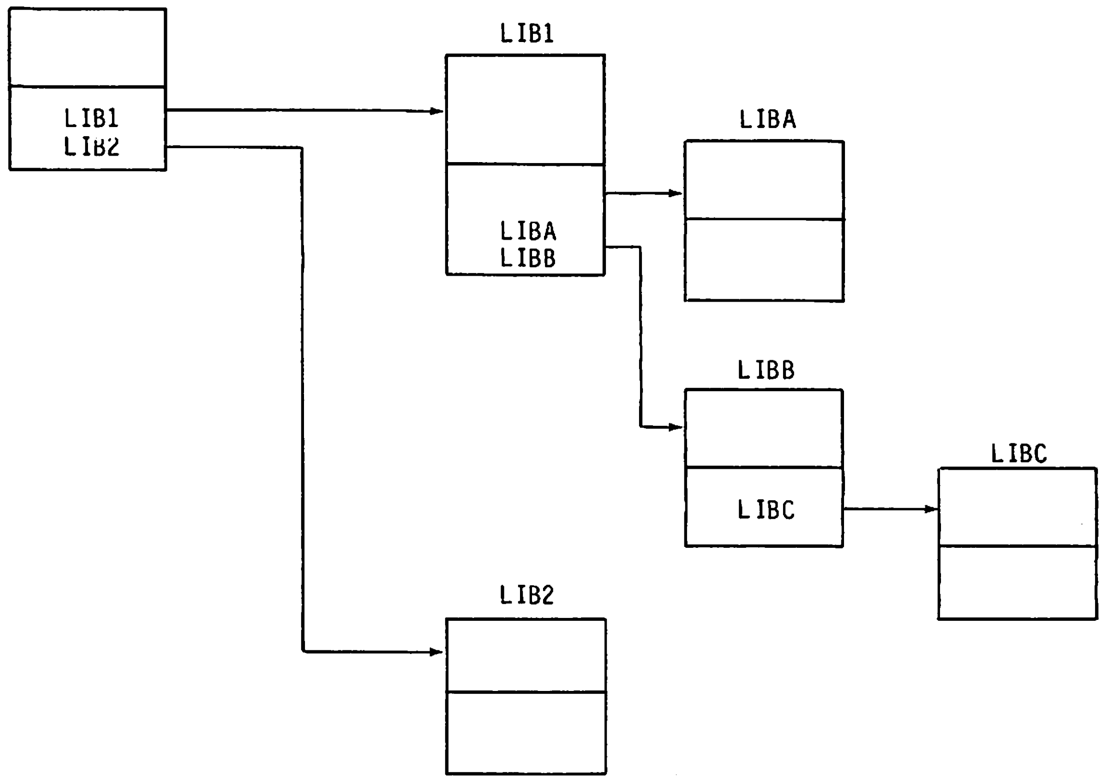
```
REMOVELIB(ABC1,ERCC01.TSSLIB)
REMOVELIB(TESTLIB)
```

The command will fail only if an attempt is made to remove the name of a library index which is not in the list.

ORDER OF SEARCHING LIBRARY INDEXES

When searching for an entry the loader first searches the current library index. If the entry is not found it then searches the appended library indexes, in the order in which they were appended. If any of these library indexes have library indexes appended then these are searched before going on to the next library index in the original append list. The diagram below shows an example of this:

CURRENT LIBRARY INDEX



The search order would be:

Current library index, LIB1, LIBA, LIBB, LIBC, LIB2

Knowledge of this order of searching is of importance when more than one entry of the same name exists in different object files (such object files cannot of course be referenced in the same library index). It enables the user to determine which object file will be loaded.

The command LIBANAL should be used to examine the contents of a library index file (see below).

Failures during loading

The following faults can occur during loading:

* Failure to find an entry. If this entry is the main entry point of a program being accessed by RUN, or is a routine being loaded as a Subsystem command then this fault causes a return to command level. Otherwise an error message is produced and the program is allowed to execute. A failure will occur if a call on the missing routine is encountered during execution of the program.

* Inconsistent library index. This failure will occur if a library index contains information about the entries in an object file inconsistent with the actual contents of the object file. This fault causes immediate return to command level.

* Too many COMMONS. 128 separate named commons or external data references have been processed, and an attempt has been made to process another one.

* Inconsistent length for COMMON. This fault occurs if a named common is initialised in a BLOCK DATA statement with a length less than one of the other references to the same common block.

* Cannot extend GLA. The space currently available for the GLA (a file called SS#GLA) is 12 segments. The GLA is created with 1 segment and extended as required. This is normally enough; although it could be extended it should be appreciated that programs requiring a data area of this size are likely to have bad paging characteristics (see chapter 1). This fault could also occur if the user's file index were full.

* Too many library index searches. This fault will occur if the library index structure has more than 16 levels - the most likely cause being a loop in the structure; e.g. A appended to B appended to C appended to A.

Loading standard commands

Commands in the standard command list (see chapter 6) are used very frequently, and in order to avoid the overhead of searching the library indexes in the user's library structure every time a standard command is used, the following action takes place:

If 'QUICKSEARCH' is selected (see OPTION in chapter 21) then whenever a command is read by the command interpreter it searches the list of standard commands before calling the loader. If the command is among those in the standard list then it must be in the Subsystem basefile, which is always loaded; a jump is then made direct to the required command. If the command is not in the standard list the loader is called to locate it in the normal way.

If the user wishes to retain the full generality of providing commands with the same names as those in the standard list he can do so by selecting the OPTION 'FULLSEARCH'. In this case all commands called are loaded by the normal method. This includes searching the user's own library structure first. There is a delay associated with this, and most users have chosen to accept the slight loss of generality which goes with the more rapid initiation of standard commands. Note that QUICKSEARCH does not apply to commands called from within programs (see chapter 18).

QUICKSEARCH is the default option.

DYNAMIC LOADING

In some cases the process of satisfying all external references before starting execution involves unnecessary work. If, during any one run, a program accesses only a few of the external routines it references then it may be better to use dynamic loading. In this case the program is loaded and allowed to start executing, and no attempt is made to locate or load any external routines. As soon as a call is made on an external routine the execution is interrupted and the loader is called to load the required routine. Execution continues until the next call is made on a routine that has not been loaded and the process is repeated.

The method of creating object files which include references to be satisfied dynamically is described in chapters 15 and 16.

Notes

* There is an overhead involved in making the additional calls on the loader. Once a routine has been loaded, however, there is no difference in the time taken for each call on that routine.

* There is little point in using this facility in a program that uses all the routines it calls. It will have the effect of making the program start execution more quickly but causing execution to be interrupted periodically to satisfy additional references.

The command LIBANAL

This command is used to examine a library index file. It takes two optional parameters. The first should be the name of a library index file, by default the user's current library index (see above). The second is used to specify a file or device to be used for the output; the interactive terminal is used if this parameter is omitted. The output is in two parts:

* A list of the object files referenced by this library index, together with the program and data entries in each.

* A list of other library index files which have been appended to this one, in the order in which they would be searched by the loader.

The command PERMITLIB

This command is used to set access permissions for a library index file. It takes parameters as for PERMITFILE (see chapter 7).

Notes

* The access permissions are applied to the library index file itself and to all the object files currently inserted in it which belong to its owner.

* Any object files subsequently inserted in the permitted library index file will have the same access permissions automatically given to them.

CHAPTER 11
DATA FILE HANDLING

This chapter introduces the subject of manipulating data in files on EMAS. It describes the structure of the two types of file used most frequently for this purpose:

* Character files

* Data files

In additionally it gives details of the following commands:

* DEFINE - links a logical I/O channel in a program to a particular file

* CLEAR - clears one or more links set by DEFINE

* DDLIST - lists current file definitions

Chapters 15, 16 and 17 describe the handling of data within the programming languages IMP, FORTRAN and ALGOL respectively. Chapter 13 describes data manipulation via direct mapping of files, and chapter 14 the use of user magnetic tapes.

CHARACTER FILES

Character files are used widely to hold textual information. They are created in the following situations:

* When the EDIT command is used - when creating a new file or as a result of use of the Editor command F (chapter 12)

* From cards or paper tape read in, other than those defined with the BINARY option (chapter 4)

* When using STREAM output from an IMP or ALGOL program

* When compiler listing files are generated

* As optional output files from commands; for example, FINDFILE(,OUT) produces a Character file OUT

* As output from the CONCAT command

The information held in a character file consists of a header, which contains length and file type information, followed by a sequence of data characters with no record separators or other system control information. For example, if a file contains the text

```
FIRST LINE
LAST LINE
```

in its only two lines then the length of the file would be the length of the header plus 21 characters for the data, a newline character following the word LINE in both occurrences. The newline characters are part of the file, and when the file is read they are used to divide it into records. There is no restriction on line length imposed by the structure of the file, but for reading using READSYMBOL in IMP a line should contain a maximum of 160 characters.

Character files provide efficient storage in that they do not have to contain trailing spaces, as do, for example, card image files used on some other systems. Furthermore, they contain no record separators, other than the newline characters themselves. They have the restriction that to be meaningful they can only be used to contain character information. If binary information were stored in a character file then it would be impossible to distinguish between newline (internal code 10) and the binary value 10 as part of a record.

Character files as input

Character files can be used as input in the following situations:

* as STREAM input to IMP and ALGOL programs; this includes source file input to the IMP, FORTRAN and ALGOL compilers, since they are IMP programs
* as input to FORTRAN programs, when using the READ statement under FORMAT control
* as EDIT input files - both for editing and for insertion using the I<filename> facility in the Editor
* as input to the commands CONCAT, DETACH and OBEYFILE

DATA FILES

Data files are distinct from character files in that they are divided into discrete records, in a way that makes it possible to store any information in them; for example, textual or binary information. They are created in the following ways:

* as output files from FORTRAN programs, written with or without FORMAT control
* as IMP Sequential or Direct Access binary files created by OPENSQ or OPENDA
* as files read in BINARY mode from cards or paper tape

Format of Data files

Data files can have either fixed (F) length or variable (V) length records. A fixed format data file consists of a header followed by one or more records with no separators between them. For some forms of data they provide a very efficient form of storage in that there is no redundant record separator information. On the other hand, for variable-length lines of text, for example, they are inefficient because they contain redundant trailing spaces.

Variable format data files contain records which are separated by control information. A file consists of a header followed by one or more records.

Apart from the user data each record contains 6 or 7 bytes of control information. For files which have long records this is not important but for files with short records this can consitute a considerable overhead on the size of the file. This should be seen in perspective however. For a file of a few pages this is not usually important. It is for applications involving large files that it is worth considering making changes, such as:

* increasing record lengths such that the control information becomes a small percentage of the whole file
* using fixed record length files

Data files for input

Data files can be used as input for FORTRAN programs and for binary sequential and direct access files in IMP. Additionally they can be used in all situations where character files are used as input, subject to the following:

* They should only contain valid character codes (see Appendix 2)
* A newline character is generated at the end of each record. However if a newline character appears within a record, it and any characters following it in the record will be ignored

Print control characters

There is an option in FORTRAN whereby the first character of the user data in a record is used to control the line spacing of the output device - for example, line printer or interactive terminal. This option can be selected in defining a file by appending an A to the record format. Hence F and V become FA and VA. When output is directed to the line printer or interactive terminal this option is selected by default. The user need only be concerned about it when writing output to a file for subsequent listing on a line printer. In this case the file should be written with a record format of FA or VA. See also 'Sequential Output', in chapter 16.

THE DEFINE COMMAND

This command is used to establish a link between a logical Input/Output channel and a particular file or output device. The command takes the following parameters:

```
DEFINE(ddname,file/dev[,size][,record format and length])
```

The DEFINE parameter: ddname

This parameter is used to determine the access method and logical channel number of the definition. The valid access methods are shown in the table below.

Valid Access Methods

Type	Allowed abbreviation	Use
STREAM	ST	IMP stream I/O, ALGOL I/O
SQFILE	SQ	IMP sequential binary file I/O
DAFILE	DA	IMP direct access binary file I/O
FT		FORTRAN file I/O
SMFILE	SM	Mapped file handling

The channel number should be a one or two digit integer in the range 1-80. Note that only one definition can exist for a particular channel number. Hence definitions for SQFILE27 and STREAM27 cannot exist at the same time. If a channel number specified in a DEFINE command has already been defined, then the previous definition is lost. Examples of valid ddnames:

```
SQFILE36

ST1

FT07
```

The DEFINE parameter: file/dev

This parameter is used to nominate the file or output device to be used. In its simplest form it can be a filename; for example:

```
DEFINE(ST1,DATA0576)
```

The file can belong to another user if it has been permitted to this user. This facility is normally restricted to files used for input:

```
DEFINE(SQ18,ERCC28.TRIAL)
```

The various output devices available are described in chapter 4. If an output device is used it should be specified with the appropriate mnemonic; for example:

```
DEFINE(FT37,.LP)
```

The interactive terminal can be used as an input or output device, and is defined thus:

```
DEFINE(ST8,.TT)
```

If it is required to concatenate input files they they should be connected with '+' characters; for example:

```
DEFINE(SQ18,FILE1+FILE2+FILE3)
```

The files used should all be of the same type and, in the case of DATA files, should all have the same record format (see earlier in this chapter).

The existence of a file can be checked by the DEFINE command, by appending -NEW or -OLD to the filename. The command will fail if the attribute given is incorrect. For example the command:

```
DEFINE(ST27,TESTOUT-NEW)
```

would fail if TESTOUT already exists. Conversely the command:

```
DEFINE(ST26,TESTIN-OLD)
```

would fail if the file TESTIN did not exist.

The qualifier -MOD can be used when defining an output file when it is required to write additional data to the end of an existing file. This is ignored if the file does not exist or is empty.

Temporary and dummy file definitions

The keyword '.TEMP' can be used instead of a file name. The effect is to generate a file definition for a temporary file. The file will be created when a program is run which sends output to the defined channel. The file will remain in existence whilst its definition is valid - that is until the command CLEAR is used (see below), or DEFINE is used again for the same logical channel.

In the following example, the program CREATE is used to write data to stream 3 (it contains a SELECTOUTPUT(3) statement), and program VALIDATE reads from stream 3 (it contains a SELECTINPUT(3) statement).

```
DEFINE(ST3,.TEMP)
RUN(CREATE)
    .
    .
    .
RUN(VALIDATE)
    .
    .
    .
DEFINE(SQ3,STORECOP)
    .
    .
    .
```

The temporary file is created when the first program is run. It is read by the second program, and destroyed when channel 3 is redefined for use with OPENSQ etc. In any event .TEMP files are destroyed at the end of a session. More than one .TEMP file can exist at a time. Each is associated with a particular channel number.

The alternative keyword '.NULL' can be used whilst testing programs. It has the following effects:

* On input - gives input ended when first accessed
* On output - all output directed to it is lost

Note that .NULL cannot be used in the case of Direct Access files, either for IMP or FORTRAN.

The DEFINE parameter: size

This parameter can be used to control the size of an output file. It should be an integer in the range 1-1023 and defines the size of the file in units of 1 Kbyte (1024 bytes). Its precise effect varies depending on the access method being used and on whether the file is new or old. The table below summarises the effect:

DDNAME	Effect of size parameter	
	New File	Old File
STREAM	determines maximum size of file for duration of current definition	determines maximum size of file for duration of current definition
SQFILE and FT (Sequential)	determines maximum size of file until it is destroyed	ignored
DAFILE	determines actual size of file	ignored
FT (Direct access)	ignored - size extracted from DEFINE FILE statement in FORTRAN program	ignored
SMFILE	ignored	ignored

Notes

* The default value for size is 255 (Kbytes).
* The size parameter should be used when directing output to a device, for example the line printer, if more than 255 Kbytes of data are being sent. For example:

  ```
  DEFINE(ST18,.LP,500)
  ```

  Note however that there are additional limitations for certain output devices; see chapter 4.

The DEFINE parameter: record format and length

The fourth parameter to DEFINE can be used to set the record format and length for an output file. It is only relevant for some access methods (see table below), and is ignored if the file already exists.

DDNAME	Record information taken from DEFINE	Note
STREAM	No	Character files do not have record format
SQFILE	Yes	
DAFILE	No	IMP DA files always have fixed 1024 byte records
FT (Sequential)	Yes	
FT (Direct Access)	No	Information taken from FORTRAN DEFINE FILE statement
SMFILE	No	

The parameter is in two parts - a format and a length. The format can be one of the following:

Format	Meaning
F	Fixed length
FA	Fixed length; first character used for carriage control
V	Variable length
VA	Variable length; first character used for carriage control

The use of the carriage control characters is explained in chapter 16.

The record length is specified in bytes. In the case of fixed length records it specifies the number of bytes in each record; hence

```
F80
```

is used for 80-byte records.

In the case of variable format records, the length includes the four bytes of record control information at the beginning of the record, and is a maximum record length. Hence if a program generates records, each containing up to 200 bytes, the user could use a format

```
V204
```

In fact automatic record spanning is used, and so if a longer record were written it would be divided into more than one record on output and then re-created as one record when it was read back in.

Notes

* The default record format for files is V1024. This can be used for almost all applications. It is rarely necessary to specify this parameter at all.

* Certain output devices impose other formats automatically - see the table in chapter 4.

  If files are written for listing on one of these devices at a later stage, the correct record format and length must be specified. For example, if creating a file for sending at a later stage to the column binary card punch, it should be defined as, say

  ```
  DEFINE(SQ27,BCPOUT,,F160)
  ```

  However if it is going straight to the column binary card punch it is not necessary to use the fourth parameter:

  ```
  DEFINE(SQ27,.BCP)
  ```

Summary of DEFINE parameters

Parameter	Position	Default	Contents	Examples
ddname	1	None	I/O type and channel no	STREAM3
file/dev	2	None	filename	ERCC27.HELP15 ABCTE
			device	.CP .GP29
			concatenated file	ABC+TEST37+END
			temporary file	.TEMP
			dummy file	.NULL
size	3	255	file size in Kbytes	500
record format and length	4	V1024	record format code and record length	F80 VA133

The command CLEAR

This command is used to clear one or more file definitions that have been established using DEFINE. It can be used in one of three ways:

* With no parameter, in which case all current definitions are cleared.

* With a list of ddnames, in which case the selected definitions are cleared.

* With a list of group names from the following list, in which case all definitions in the selected groups are cleared:

  'STREAMS', 'SQFILES', 'DAFILES', 'FTFILES', 'SMFILES'.

Examples:

```
CLEAR
CLEAR(ST1,STREAM27,DAFILE42)
CLEAR(SQFILES,DAFILES)
```

Notes

* All definitions are cleared automatically at the end of a foreground session.

* If a DEFINE is used for a logical channel for which there is already a definition, the earlier definition is automatically cleared.

The command DDLIST

This command is used to provide a list of current file definitions. It can be used without a parameter, in which case output goes to the interactive terminal, or with a parameter to specify an output device or file:

```
DDLIST
DDLIST(.LP)
DDLIST(DDFILE)
```

Typical output from DDLIST:

```
SQFILE01    TESTSQ
DAFILE07    TRYPACK
FT14        .LP
```

CHAPTER 12
THE SUBSYSTEM EDITOR

The standard Subsystem includes a context editor which is invoked by a call of the command EDIT. This chapter describes the EDIT command and the command language used to control the editor. At the end of the chapter there is a section describing the commands LOOK and RECALL, which use a subset of the editor command language.

THE EDIT COMMAND

This command can be used to examine or alter the contents of a character file. There are four ways in which the EDIT command can be used:

* EDIT (newfile) - in this case 'newfile' is the name of a file that does not currently exist. The editor will create a file with the name 'newfile' and insert text as instructed by the use of appropriate editor commands. A confirmatory message 'newfile IS A NEW FILE' will be printed.
* EDIT (oldfile) - in this case the file 'oldfile' does already exist. The effect is to make changes in the file 'oldfile' according to the editor commands used.
* EDIT (oldfile,newversion) - in this case the 'oldfile' will be copied into 'newversion' and will not be altered itself. Editor commands used will alter the copy in 'newversion'. Note that if 'newversion' does not exist it will be created, and if it does it will be overwritten.
* EDIT (oldfile,.NULL) - see 'LOOK' at the end of this chapter.

Method of editing

Editing is accomplished by moving a cursor through the file and inserting and removing text with respect to the current position of the cursor. On entry to the editor the cursor is positioned at the top (beginning) of the file.

COMMAND STRUCTURE

All editor commands are single letters. In some cases they are followed, immediately, by one of the following:

* An integer which must be typed as a sequence of numeric characters optionally preceded by a minus sign.
* A text string which is a sequence of any characters (including newline) delimited by a pair of one of the following characters: / . ?, optionally preceded by a minus sign.

  Examples:
  ```
      /ABC/   ?12*23(A/27)?   .IS
      THIS.
  ```

  Note that if the delimiter character appears in the text it must be typed twice in order to distinguish it from the closing delimiter.

  Example:
  ```
      .A=2..3*(B/PI).
  ```

* A filename which should be enclosed in the characters '<' and '>'.
  Example:
  ```
  <MYFILE>
  <ERCC06.EDITTEST>
  ```

* The single character ' (quote). This is used to indicate the same text string as that last used in the use of this command. Hence the sequence TM/%END/M1M' has the effect of moving the cursor to the second occurrence of the text '%END'.

There is a table later in the chapter which indicates the valid parameter types for each command. Commands can be typed one to a line, or concatenated, without separators, on a line.
Example:

```
TP10M/%ENDOF/
```

Note that a failure in a command within the sequence will result in the termination of the sequence, at the point of the failure. Spaces within commands, apart from those in text strings, are ignored - hence the following example would have the same effect as the last one:

```
T P10 M /%ENDOF/
```

Commands used to alter the position of the cursor

The following commands are used to move the cursor around within the file in preparation for inspecting the contents or altering some part of it. They do not alter the contents of the file.

T - Top. This command takes no parameter. It moves the cursor to the top (beginning) of the file.

B - Bottom. This command takes no parameter. It moves the cursor to the bottom (end) of the file.

M - Move. This command can be used with an integer parameter, in which case the effect is to move the cursor from its present position the number of lines specified by the parameter and position it at the beginning of the selected line. Thus M4 means move down the file four lines.
M-1 means move to the beginning of the previous line.
M0 means move to the beginning of the current line - i.e. the line currently containing the cursor. If there are insufficient lines in the file the cursor is left at the bottom or top of the file, depending on the sign of the parameter.

Move can also be used with a text string, in which case the cursor is moved from its present position down the file to the beginning of the specified text string. Hence M/%END/ means move the cursor from its present position to the beginning of the next occurrence of the text '%END', and M/4/ means move to the first occurrence of the character '4'. Note the difference between M4 and M/4/. If the text is not found the cursor is left at the bottom of the file. If the delimited text is preceded by '-' the effect is to move up the file to the beginning of the required text. If the text is not found the cursor is left at the top of the file.
Example:

```
M-/%BEGIN/
```

A - After. This command used with an integer parameter alters the position of the cursor the number of characters specified by the value of the parameter. Hence A3 means move the cursor three characters down the file and A-3 means move the cursor back three characters. All characters in the text are counted, including newline.

If there are insufficient characters in the file to allow the command to complete, the cursor is left at the bottom or top of the file, depending on the sign of the parameter.

The command A can also be used with a text string, in which case it has an effect similar to Move, with the difference that the cursor is moved to after the first occurrence of the specified text. Hence

```
A/%ENDOFPROGRAM/
```

would move the cursor to after the 'M' of '%ENDOFPROGRAM'. If the text is not found the pointer is left at the bottom of the file. As with MOVE the delimited text can be preceded by '-' to cause the search to move up the file from the present position.

G - Go to character. This command is used with an integer parameter and has the effect of moving the cursor to before the character position on the current line specified by the value of the parameter. If the requested position is beyond the end of the current line the line is extended with space characters up to the cursor. Hence if it is required to put a comment starting at column 40 in an IMP source file, the command G40 would position the cursor correctly for making the insertion. If the value of the parameter is less than 1 it is treated as 1 and if greater than 132 it is treated as 132.

H - Hold. This command is used to move the cursor to the position it was in at the beginning of the last sequence of commands. This is particularly useful when a mistake is made in typing a text parameter.
Example:

```
EDIT: M/%EMD/
*B*
EDIT: HM/%END/
%END
```

Command used to insert text

The command I is used to insert text immediately before the present position of the cursor. When used with a text string, this is the text to be inserted. Hence I/276/ would insert the text '276' immediately before the present position of the cursor.

Alternatively 'I' can be used with a filename parameter, in which case the contents of the specified file are inserted before the current position of the cursor:

```
I<HEADFILE>
```

The file 'HEADFILE' is not altered by this operation.

Deletion of text

Two commands are provided for deleting text:

D - Delete. This can be used with an integer parameter to delete the number of lines specified by the value of the parameter. For a positive value the lines are deleted from the current line down the file, and for a negative value the lines preceding, but not including, the current line are deleted. Hence:

D1	delete the current line
D3	delete the current and two following lines
D-7	delete the seven lines immediately before the current line

If an attempt is made to delete more lines than exist before or after the pointer, the lines that do exist are deleted.

The command D can also be used with a text string, in which case the effect is to delete all the text from the current position of the cursor up to and including the first occurrence of the specified text. Hence in a textual file the sequence:

```
M/There/D/./
```

would have the effect of deleting the whole of the first sentence beginning with 'There' after the present position of the cursor.

After use of the D command the cursor is left in the position previously occupied by the deleted text. If the specified text is not found no text is deleted and the pointer is left at the bottom of the file.

R - Remove. This is used with an integer parameter to remove a specified number of characters, from the present position of the cursor. Hence R3 means remove the 3 characters immediately after the cursor and R-10 means remove the 10 characters immediately before the cursor.

If an attempt is made to 'Remove' more characters than exist before or after the cursor the characters that do exist are removed.

'R' can also be used with a text string, in which case the effect is to remove the first occurrence of the specified text, after the present position of the cursor. Hence M/There/R/./ would have the effect of removing only the '.' following the first occurrence of 'There'. Note carefully the distinction between D/./ and R/./.

After use of 'R' the cursor is left in the position previously occupied by the deleted text. If the specified text is not found the cursor is left at the bottom of the file.

Terminal output from the editor

After each command or each sequence of concatenated commands, the current line (the line containing the cursor) is listed on the terminal. Alternatively the editor command P can be used to Print lines.

P can take an integer parameter, in which case the effect is to print the specified number of lines - P10 means print 10 lines starting at the current line and P-10 means print the 10 lines before the current line and the current line.

P can take a text string parameter, in which case printing starts at the current line and goes on to the line containing the first occurrence of the specified text.

Note that the cursor is not moved by the 'P' command.

The cancel command

If an error is made in typing editor commands the normal rules apply for deleting characters or a whole line (see chapter 5). Additionally the editor command 'C' can be used to Cancel editor commands in the current command string. 'C' must be followed by an integer to specify the number of editor commands before the 'C' to be cancelled. For example:

```
TI/TEXT HERE/M27C2I/** TEXT/M20
```

This would have the effect:

```
TI/** TEXT/M20
```

If the value of the integer is greater than the number of commands in the current string then they are all cancelled. It is not possible to cancel the effect of commands in command strings that have already been completed.

If command repetition is used (see below) it should be noted that, when calculating the parameter for 'C', a closing bracket and the integer that follows it are counted as one command and that opening brackets are not counted.

Terminating an edit session

There are two commands for terminating editing:

E is used with no parameter as the normal exit command. The effect is to return to normal Subsystem command level.

Q is used as an exit when for some reason the editing done during a session is not required. The effect is to leave the file or files being edited in the state that they were in before use of the EDIT command. In order to reduce the risk of a user inadvertently pressing Q and losing his editing unintentionally this command does not cause an immediate exit but causes the prompt 'QUIT': to appear, to which the user should reply 'Q' or 'Y' if he really does want to exit. Any other reply will result in the edit session continuing.

MORE ADVANCED FACILITIES IN THE EDITOR

The commands described so far provide most of the commonly required functions of a context editor. There are three further facilities described here which may be of interest to some users:

* repeated commands
* moving a section of text within a file
* extracting part of a file and putting it into another file

Command repetition

A single editor command or group of commands can be obeyed repeatedly a specified number of times. The commands are enclosed in parentheses followed by an integer repetition factor. For example, if it is required to remove all occurrences of the text 'REAL' in a file and to replace them with the text 'INTEGER' one could type

```
(R/REAL/I/INTEGER/)1000
```

This assumes that there are not more than 1000 occurrences of the text 'REAL'. Another use of this facility might be to find out the names of the next 10 subroutines in a Fortran program. The command sequence to do this would be:

```
(M/SUBROUTINE/P1M1)10
```

Note that strictly this example would print the next 10 lines that contained the text 'SUBROUTINE'. Since this word might appear in a comment the command sequence might not achieve the required effect. Note also the 'M1' in the command sequence. If this were not included the effect would be to print the next line containing 'SUBROUTINE' ten times. This example illustrates the need to consider carefully the effect of repetitive editing commands.

Bracketed commands may be nested.

The normal rules concerning failures within commands are followed. If a failure occurs the whole sequence of commands is aborted.

The separator *S*

The editor command 'S' is used to set a separator before the present position of the cursor. This separator is used to indicate the destination of text being moved (see below). Additionally it has the effect of a separator in the file. Three commands can be used to move the cursor past the separator:

* T. This moves the cursor to the top of the file, if necessary passing the separator
* B. This moves the cursor to the bottom of the file, if necessary passing the separator
* O (Over). This command, which takes no parameter, moves the cursor to immediately before the separator. This is a more efficient operation than, for example, TM1000, which would be the form needed if the cursor were positioned after the separator.

All other commands which move the cursor can only move it as far as the separator. This fact can be utilised when it is required to search only part of a file for text strings. Before starting, the user positions the separator at the end of the text to be searched. The position of the separator is indicated by the text

```
      *S*
```

This text does not actually exist in the file.

The separator can be moved from the file by use of the command 'K'. This command takes no parameter and leaves the cursor at the point previously occupied by the separator.

Moving a section of text within a file

The process of moving part of a file from one place to another within the file involves

* setting a separator at the destination of the file, using the command 'S'
* moving the cursor to the top (start) of the text to be moved
* using the command 'U' to move the text
* removing the separator with the command 'K'

The commands 'S' and 'K' are described above. The command 'U' takes either an integer parameter or a text string parameter. When used with an integer the value of the parameter specifies the number of lines to be moved, counted from the current line. When a text string is used the text moved extends from the current position of the cursor up to and including the first occurrence of the specified text. In the following example the routine B is to be moved to before routine A.

Initial state of file:

```
*T*
%BEGIN
%ROUTINE A
    TEXT OF ROUTINE A
%END
%ROUTINE B
    TEXT OF ROUTINE B
%END
    TEXT OF PROGRAM
%ENDOFPROGRAM
*B*
```

```
EDIT:M/%ROUTINE A/S          Set separator
*S*
↑%ROUTINE A
EDIT:M/%ROUTINE B/U/%END     Move text
/:/
↑   TEXT OF PROGRAM
EDIT:K                       clear separator
↑%ROUTINE A
```

Final state of file

```
%BEGIN
%ROUTINE B
    TEXT OF ROUTINE B
%END
%ROUTINE A
    TEXT OF ROUTINE A
%END
    TEXT OF PROGRAM
%ENDOFPROGRAM
```

Extracting part of a file

The command 'F' is used to extract part of a file and to put it into another file or send it to an output device. The parameter must be of the form

```
       <filename>
or     <output device>
```

The abbreviations used for output devices are given in chapter 4. The text which is extracted is that which lies between the present position of the cursor and the 'S' separator, if it is positioned lower in the file than the cursor; otherwise, between the cursor and the bottom of the file. The cursor is not moved by this command and the text in the file being edited is not altered. If the parameter specifies a file that already exists, the file will be over-written; otherwise a new file will be created. This command can be used, for example, for extracting one routine from a file for use in another program, or for listing a part of a long file on the line printer. In the previous example the %ROUTINE B could be listed on the line printer using the following sequence of commands:

```
EDIT:M/%ROUTINE B/M/%END/M1S       Set separator at bottom of routine
*S*
↑%ROUTINE A
EDIT:OM-/%ROUTINE B/F<.LP>         List routine on line printer
↑%ROUTINE B
```

THE OPERATION OF THE EDITOR

Although it is possible to use the editor with no knowledge of its internal workings some users might appreciate a brief description. The editor makes use of the virtual memory and handles its files by directly addressing them (see chapter 1). When editing one file to another it first connects the file in the virtual memory and sets up pointers to the top and bottom. Any text that is inserted is stored in a work area in the virtual memory and each section of text has pointers to the top and bottom. Any operation which divides a section of text - for example removing a character from the middle of it, results in additional pointers being set up pointing to the beginning and end of the 'hole'. All these pointers are linked together in the logical order in which the sections they point to appear in the file. Note that this order may bear no resemblance to the order in which the sections are laid out in the store. When the edit command 'E' is reached, an output file is created, if necessary, and the sections of text are moved into it in the correct order, as determined from the linked list of pointers.

Note:

* Since the output file is not constructed until the E command is executed all editing is lost if a system failure occurs during editing.

* Since there are pointers to the top and the bottom, moving the cursor to these points with T and B is efficient; thus when moving from the top of a long file to a point near the bottom it is more efficient to do

  ```
  BM-/TEXT/      than
  M/TEXT/
  ```

  When relevant O is also an efficient command.

The command LOOK

The command LOOK is used to activate the editor for the purpose of examining, rather than altering a file. It takes one parameter - the name of the file to be examined - with a default of 'SS#LIST', the default compiler listing file (see chapter 9).

A similar effect can be achieved by typing

```
EDIT (filename,.NULL)
```

There are two differences between using LOOK and EDIT:

* the editor commands I, R, D, U, and G are not allowed since they alter the file
* the prompt at editor command level is 'LOOK:'

Otherwise the facilities available are identical.

The command RECALL

This command is used to interrogate the file containing a copy of all interactive terminal Input/Output operations for this user; see also chapter 21. It takes no parameter. There are three differences between RECALL and EDIT:

* the editor commands I, R, D, U and G are not allowed since they alter the file
* on entry the cursor is at the bottom of the file - i.e. pointing at the end of the most recent information
* the prompt at editor command level is 'RECALL:'

The table below shows the available commands and the parameter types that can be used with each. The final column shows which commands can be used with LOOK and RECALL.

Valid Parameters								
Command	Use	None	Integer	Text string	- Text string	Quote	Filename	Allowed in LOOK and RECALL
A	After		*	*	*	*		*
B	Bottom	*						*
C	Cancel		*					*
D	Delete		*	*		*		
E	Exit	*						*
F	File						*	*
G	Go to		*					
H	Hold	*						*
I	Insert			*		*	*	
K	Kill	*						*
M	Move		*	*	*	*		*
O	Over	*						*
P	Print		*	*		*		*
Q	Quit	*						*
R	Remove		*	*		*		
S	Separate	*						*
T	Top	*						*
U	Use		*	*		*		

CHAPTER 13
STORE MAPPING

Accessing data by direct mapping

This chapter describes the facilities provided for accessing the contents of files by mapping them onto data structures in IMP programs FORTRAN and ALGOL users can make use of these facilities most easily by writing an interface routine in IMP (see references 3 and 9).

Principle of operation

As explained in chapter 3, all files (except those held on magnetic tape) are accessed by connecting them at an address in a very large virtual memory. The supported languages include routines such as READSQ to enable user programs to access the contents of files in a conventional manner and this provision expedites the transfer of programs to and from EMAS. On the other hand these routines are unnecessarily inefficient, in that each character that is accessed has to be moved from one address in the virtual memory (within the file) to another address in the virtual memory (within the user program's data area). The facilities described in this chapter remove the requirement for this intermediate movement of the data, and at the same time free the user from the restraints imposed by file formats; for example, the restriction that IMP direct access files must have 1024 bytes per block.

Direct mapping can be used most readily with IMP mapping facilities and the user is referred to the IMP manual (reference 1) for further information. In particular the use of array pointer variables is used in the following examples. Alternatively record pointer variables could be used.

File types suitable for direct mapping

Any file type can be used for direct mapping but three types are especially suitable:

* Character files: these have a very simple structure.
* Data files with fixed length records: the use of data files with variable length records is not recommended, since the precise format of the record separators is likely to change.
* Store Map files: these are files which are created especially for direct mapping. They have no pre-defined structure, and can only be used for this type of file access.

Creating Store Map Files

The command NEWSMFILE is used to create a store map file. It takes two parameters - the name of the file to be created and the length of user data required (in bytes). For example, if it were required to create a file to hold an array of 1000 longreal variables, one could give the command

```
NEWSMFILE(STORE,8000)
```

Linking the file to a data structure within a program

The DEFINE command is used to link a particular file to a logical channel for use in the program. In this case only the first two parameters are relevant. For the above file one could specify

```
       DEFINE(SM27,STORE)
```

The use of DEFINE makes it possible to re-run the program with different sets of data, without having to modify it.

Within the program it is necessary to include a call on the integer function SMADDR to connect the file and determine the address in the virtual memory at which it is connected. The routine should be specified thus:

```
       %EXTERNALINTEGERFNSPEC SMADDR(%INTEGER CHAN, %INTEGERNAME LEN)
```

In order to obtain the address of the first byte in the file STORE used above, one could include the statements:

```
       %INTEGER START,LEN
            .
            .
            .
            START=SMADDR(27,LEN)
```

The second parameter, LEN, is used to return the length of user data in the file to the program - in this case 8000. This can be useful for checking, and in order to obtain the length of an unknown file (see later example).

The individual variables in the file STORE could be accessed as

```
            LONGREAL(START)
            LONGREAL(START+8)
            LONGREAL(START+16)
            .
            .
            .
            etc.
```

It is more convenient, however, to map them onto an array. The method is to specify an array format and an array name, and to equivalence the array name to the file:

```
       %LONGREALARRAYFORMAT OUTAF(1:1000)
       %LONGREALARRAYNAME OUTA
            .
            .
            .
            .
            OUTA==ARRAY(START,OUTAF)
```

(It is assumed that START has been assigned a value using SMADDR, as shown above.) From this point on, the elements of the array can be accessed as OUTA(1) to OUTA(1000).

Effect of accessing the array

It should be understood that the array and the file are one and the same. Thus if the following code were executed the effect would be to set to 0 the first 20 elements of the array - that is the first 20 values in the file. This change does not last merely until the end of the program but is permanently recorded in the file, and any subsequent access to this file will find these elements cleared to zero:

```
       %CYCLE I=1,1,20
           OUTA(I)=0
       %REPEAT
```

Thus, where a file has to be modified as the result of a program run, it is clearly desirable to use direct mapping, since the file is thereby kept up to date at all times. This is particularly useful when the program terminates unexpectedly.

It is possible to use much more complex data structures if records are used, and the combination of record manipulation and direct file mapping provides a useful tool for applications requiring random access to large and complex directories or tables.

In the next example direct mapping is used to access the contents of a character file. The structure of a character file is described in chapter 11. This routine could be used to count the number of lines in the file - far more efficiently than by using READSYMBOL.

```
%EXTERNALINTEGERFNSPEC SMADDR(%INTEGER A, %INTEGERNAME B)
%ROUTINE COUNT LINES(%INTEGER CHAN)
%INTEGER START,FLENGTH,I,LINES
    START=SMADDR(CHAN,FLENGTH); !FLENGTH IS NO OF BYTES IN FILE
    LINES=0
    %CYCLE I=START,1,START+FLENGTH-1
        %IF BYTEINTEGER(I)=NL %THEN LINES=LINES+1
    %REPEAT
    PRINTSTRING('NUMBER OF LINES IN FILE = ')
    WRITE(LINES,1)
    NEWLINE
%END
```

In the following example a program has been written to print out the numerical value of the bytes in a given record in a file with fixed length records which has been DEFINEd as SM80:

```
        %BEGIN
        %EXTERNALINTEGERFNSPEC SMADDR(%INTEGER CHAN, %INTEGERNAME LEN)
        %EXTERNALROUTINESPEC PROMPT(%STRING(15) S)
        %INTEGER RECL, RECNO, START, LEN, MAX, RECSTART
            START=SMADDR(80,LEN)
            PROMPT('RECORD LEN:')
            READ(RECL); !LENGTH OF EACH RECORD
            MAX=LEN//RECL; !MAX IS HIGHEST RECORD NO
            PROMPT('RECORD NO:')
NEXT:       READ(RECNO)
            %STOP %IF RECNO<1; !END RUN
            %IF RECNO>MAX %THENSTART
                PRINTSTRING('BEYOND END OF FILE')
                NEWLINE
                -> NEXT
            %FINISH
            REC START=START+RECL*(RECNO-1); !START OF REQUIRED RECORD
            %BEGIN; !NEW BLOCK TO ALLOW DYNAMIC DEFINITION OF BAF
            %BYTEINTEGERARRAYFORMAT BAF(1:RECL)
            %BYTEINTEGERARRAYNAME BA
            %INTEGER I
                BA==ARRAY(RECSTART,BAF)
                %CYCLE I=1,1,RECL
                    WRITE(BA(I),3)
                    NEWLINE %IF I&X'F'=0; !NEWLINE EVERY 16 NOS.
                %REPEAT
            %END
            ->NEXT
        %ENDOFPROGRAM
```

Closing mapped files

At times it is necessary to close a mapped file. In this context closing implies removing the link between the file and the logical channel in the program. The result is to free the file for other use; for example, it might be necessary to access the file by another access method in the same program. The program must be written in such a way that after closing the file it makes no reference to the array that has been equivalenced to it. The effect of doing so is undefined but could include corruption of files other than the mapped one. This is because the addresses used for the mapped file might be re-used for another file.

The routine used for closing a file must be specified:

```
%EXTERNALROUTINESPEC CLOSESM(%INTEGER CHAN)
```

and the call would be, for example:

```
      CLOSESM(27)
```

Changing the size of a mapped file

The routine CHANGESM can be used to change the size of a mapped file. It must be specified:

```
      %EXTERNALROUTINESPEC CHANGESM(%INTEGER CHAN, NEWSIZE)
```

where

CHAN	is the channel number on which the file is defined, and
NEWSIZE	is the new size required, in bytes.

Notes

* The file must be DEFINEd but must not be OPEN - i.e. the call should be made before a call of SMADDR, or after a call of CLOSESM. In the latter case another call of SMADDR must be made after the call of CHANGESM and any mapping of an array onto the file repeated because the file may have been moved to a different location in the virtual memory when its size was changed.

* NEWSIZE can be larger or smaller than the present size. If it is smaller, then any data beyond the end of its new size is lost. Otherwise the contents of the file remain unchanged.

* It is recommended that the use of CHANGESM be restricted to store map files, i.e. those that have been created explicitly for mapping using the NEWSMFILE command.

In the next example the file 'STORE' used in an earlier example is extended to allow the program to access 1500 longreal variables; it is assumed that the command DEFINE(SM72,STORE) has already been given:

```
      %BEGIN
      %EXTERNALROUTINESPEC CHANGESM(%INTEGER CHAN,NEWSIZE)
      %EXTERNALINTEGERFNSPEC SMADDR(%INTEGER CHAN, %INTEGERNAME LEN)
      %LONGREALARRAYFORMAT OUTAF(1:1500)
      %LONGREALARRAYNAME OUTA
      %INTEGER LEN,START
      !STORE IS DEFINED ON CHANNEL 72
          CHANGESM(72,12000); !EXTEND FOR 1500 LONGREALS
          START=SMADDR(72,LEN)
          .
          OUTA==ARRAY(START,OUTAF)
          .
          .
```

Store mapping and program portability

It is important to appreciate that the facilities described in this chapter are specific to EMAS. When working on programs that are likely to be moved to other systems the user should ascertain whether similar facilities exist there. For example, the current implementation of IMP on IBM 370 series computers makes no provision for direct addressing of files. On the other hand, for programs written specifically for use on EMAS the use of direct mapping merits serious consideration.

Conclusion

One of the problems involved in using mapped files is overcoming the conceptual block to the idea that a program can access a file without calling read and write routines. For many years programs have been written that access input/output devices directly, or at least via a spooling system. At first the direct mapping of files seems to be a complicated extension - in fact it is a real simplification. The advantages of accessing data by direct mapping are:

* Simplicity of associated programming: mapping can be regarded as a method of saving data held in arrays from one use of a program to the next.

* Efficiency: the only part of a file that needs to be referenced is that part required for the current run. The paging mechanism (see chapter 3) will ensure that only the pages referenced are brought into main store. Additionally this access method avoids the overhead of copying data between a file and a user's data area.

* No limitations imposed by file formats: a mapped file can consist of one byte or many thousands of bytes linked together in a way that is convenient to the programmer.

CHAPTER 14
MAGNETIC TAPE FILE HANDLING

The primary use of magnetic tapes in the EMAS system is for the archive and back-up components of the file system, described in chapter 3. These functions are controlled by the system and the user need have no knowledge of tape formats, serial numbers and so on. The Subsystem does include a user interface to the magnetic tape handlers which is described in this chapter. There are however severe restrictions on the use that can be made of these facilities. Since there are only four tape decks for both System 4 computers (normally configured as two on each machine), it is not normally possible to allow users access to more than one tape at a time. This is because one deck is needed on each machine for restoring files. Further, the heavy use of the tape decks for file system functions imposes the limitation that at present access to the tape decks is only allowed for background jobs run overnight, and even then only for a limited number of users. This restriction on the use of private magnetic tapes should be seen in the context of a system that provides many of the functions automatically. The archive store provides the majority of users with all they need in the way of facilities for the long term storage of data.

There are, however, two groups of users who need access to magnetic tape:

* Those who wish to access large files. (Currently there is a management imposed restriction of 1 Mbyte as the maximum size of file system files; above this there is an absolute restriction of 4 Mbytes imposed by the design of EMAS.)

* Those who wish to bring to the system data that is currently stored on magnetic tape, or take data from the system to another installation on magnetic tape.

Magnetic tape hardware

The tape decks on the System 4 computers used for EMAS can read and write only magnetic tapes with the following characteristics:

No. of Tracks	9
Width	0.5 inch
Packing Density	1600 bpi
Mode	Phase Encoded
Parity	Odd

Users having tapes destined for EMAS with characteristics which differ from these should contact the Advisory Service in the first instance, where they can obtain information about converting them to a suitable form for input.

Tape labelling standard

The user tapes read and written on EMAS should have IBM OS/370 compatible labels (reference 12). There are some other restrictions:

* Only record formats F, FA, FB, FBA, V, VA, VB, VBA are allowed.

* The maximum blocksize allowed is 12288 (12 Kbytes).

* Multi-reel files are not allowed.

Users having 9-track 1600 bpi tapes not conforming to these formats should also contact the Advisory Service to obtain information about conversion.

Accessing magnetic tapes

Magnetic tape files can be accessed from IMP using the sequential binary input/output routines (SQFILES) or from FORTRAN using the standard formatted or unformatted READ and WRITE statements. They cannot be accessed from IMP as STREAMs. Instead of using the command DEFINE to establish a link between the logical channel and a particular magnetic tape file, the user should use the command DEFINEMT.

The command DEFINEMT

This command takes up to six parameters, of which the first three are essential.

```
DEFINEMT(ddname,file,vol[,label][,record format and length][,blocksize])
```

ddname	should be of the form SQFILEnn or FTnn and is as for the command DEFINE.
file	should be the file name of the file being read or to be written. It is suggested that when writing tapes normal EMAS filenames be used; for example ERCC06.TESTA However the software does allow the user to specify up to 17 characters and the name does not have to be in the standard EMAS format. This feature is needed when reading tapes written at other installations. Note however that when reading tapes only the first 15 characters of the name on the tape are checked, and when writing tapes only the first 15 characters of the given file name are used.
vol	is the 5 or 6 character tape serial number. If it is followed immediately by an asterisk the tape will be loaded with a write permit ring fitted. This is essential if it is intended to write to the tape. Otherwise no write ring will be fitted and the tape will thus be protected from inadvertent overwriting.
label	should be a positive integer and is used to specify the position of the file on the tape (default = 1).
record format and length	should be specified as for DEFINE. This parameter is ignored when reading, when the information is extracted from the tape label (default = V1024).
blocksize	should be a positive integer in the range 18-12288. It is used when writing fixed record-length files to specify the length of each block, and in the case of variable record-length files to specify the maximum length of each block. Note that this parameter and the previous one are used to determine whether the records should be blocked. For example, if the two parameters were F80 and 4000 respectively, the file would be written with 80 byte records, blocked up to 50 to a block of 4000 bytes. If, in the case of fixed records, the blocksize is not an exact multiple of the record size, it is decreased to the last exact multiple. This parameter is ignored when reading a tape - the information is instead extracted from the tape label (default = 4096).

Examples of valid calls of DEFINEMT

```
DEFINEMT(SQ7,ERCC27.TESTTAPE,AS1273)

DEFINEMT(FT18,EJNN33.DATA2704,EA1777*,7,F100,8000)

DEFINEMT(FT5,ERNN.JXTB15.17SST,X13621)
```

Restrictions on language facilities when using magnetic tape

Note that certain restrictions are imposed on the file manipulation facilities when using magnetic tape files:

* The FORTRAN BACKSPACE facility cannot be used.

* Only one file on a tape can be open at one time. In the case of IMP the routine CLOSESQ can be used to close one file before using OPENSQ for another. For FORTRAN users the language does not include an explicit close facility, and so a non-standard routine CLOSEF is provided (see chapter 16).

Character codes

Magnetic tapes containing binary information written on IBM 360 or 370 series machines can be read without conversion, so long as they conform to the limitations specified earlier in this chapter.

In the case of character information, if FORTRAN reading or writing under FORMAT control is used, the information will be translated to or from EMAS internal code from or to EBCDIC on magnetic tape. Thus a tape written using IBM or Edinburgh FORTRAN on an IBM 370 can be read without any explicit translation being required.

When reading or writing is carried out with the IMP sequential binary routines, no translation occurs.

Operational considerations

Before embarking on the use of magnetic tape facilities, the user should contact the Operations Controller to make arrangements for the following:

* storing his own magnetic tapes in the machine room

* obtaining additional magnetic tapes

* running jobs using magnetic tapes

Development

The user magnetic tape facilities are still being developed on EMAS. The Advisory Service should be contacted in the first instance for details of any recent developments.

CHAPTER 15
IMP ON EMAS

Most of the EMAS operating system and standard Subsystem is written in the high level programming language IMP. Hence EMAS provides an ideal environment for running programs written in this language. The language is described in references 1 and 11. This chapter describes, for the IMP programmer, the environment in which his program will run.

Compilation

An IMP source file can be compiled using the command IMP, as described in chapter 9. The related command PARM, used to set compile time options, is described in the same chapter.

Programs

An IMP program consists of one program block bounded by %BEGIN and %ENDOFPROGRAM. This block can contain inner blocks, routines and functions, all of which are local to the program. It can also contain references to other separately compiled routines and functions. These are specified by %EXTERNALROUTINESPEC statements etc.

When the program has been compiled the resulting object file can be executed by use of the RUN command.

If an object file produced in this way is analysed by FILEANAL it will be seen to have only one entry point, which is given the name S#GO. This name is used to avoid conflict with user-written %EXTERNAL routines, which cannot have names containing the '#' character.

%EXTERNAL routines

Instead of writing IMP as a program it can be written as a file of %EXTERNAL routines or functions. There are two uses of this facility:

* To provide separately compiled routines for calling from programs or from other %EXTERNAL routines.

* To make routines which can be executed as separate entities. This is described more fully in chapter 18.

Note the following characteristics of entry points:

* Only the first 8 characters of the name of the routine or function is used for external linking. Care should be taken to use names for external entities that differ in their first 8 characters.

* No distinction is made in the entry point list between routines, functions or maps. Nor is any information about the parameter list included with the entry point. Thus to ensure correct operation it is vital that the %SPEC statement used to define an external reference has the same type and parameter list as the routine, function or map itself. The names used for parameters can differ but their types must be the same, and they must be given in the same order. Thus a valid specification for WFILE in the following example would be

```
%EXTERNALINTEGERFNSPEC WFILE(%INTEGER C, B)
```

The source file should contain one or more %EXTERNAL routines, each terminated by %END, and the whole file should be terminated by %ENDOFFILE. Note that %EXTERNAL routines cannot be nested; on the other hand each routine can contain routines and blocks. Also the file can contain routines and functions which are global to the whole file, though not themselves %EXTERNAL. Further, it can contain references to other %EXTERNAL routines and functions, whether they are part of the file or not. An example of such a file is:

```
       %ROUTINE SETUP(%INTEGER N)
           .
           .
           .
       %END
       %EXTERNALROUTINE FILE(%INTEGER IN, START)
           .
           .
           .
           SETUP(IN)
           .
           .
           .
       %END
       %EXTERNALINTEGERFN WFILE(%INTEGER OUT, START)
       %EXTERNALROUTINESPEC WRITESQ(%INTEGER C, %NAME B,E)
           %ROUTINE CHECK
           .
           .
           .
           %END
           SETUP(OUT)
           CHECK
       %END
       %ENDOFFILE
```

If the object file produced by compiling the above source file were analysed using the command FILEANAL it would be found to contain entry points FILE and WFILE. Note that the routine SETUP is not accessible directly from outside the file. It is only executed as a result of calls on FILE or WFILE.

%EXTERNAL data

Apart from calling routines, functions and maps in separately compiled object files it is also possible to access variables declared in separately compiled object files. The declarations of variables to be used in this way must be qualified by %EXTERNAL in the file in which they appear. Note that the %EXTERNAL qualifier gives them additionally the characteristics of %OWN variables. The effect of including, for example,

```
       %EXTERNALINTEGER BASE
```

in a file is that the compiled object file will contain a data entry BASE.

In order to access an %EXTERNAL variable from another file it should be declared as an %EXTRINSIC variable in that file. The effect will be to establish a link, when the file is loaded, to the %EXTERNAL variable of the same name. In the following example the array TABLE is accessible to both routines START and CHECK even though they are contained in separate object files:

```
       %EXTERNALINTEGERARRAY TABLE(1:100)
       %EXTERNALROUTINE START
           .
           .
           .
           TABLE(1)=244
           .
           .
           .
       %END
       %ENDOFFILE
```

```
      %EXTRINSICINTEGERARRAY TABLE(1:100)
      %EXTERNALROUTINE START(%INTEGER I)
          .
          .
          .
          %IF TABLE(I)=244 %START
          .
          .
          .
      %END
      %ENDOFFILE
```

Notes

* As with routine entries only the first 8 characters of the name are significant.

* The name, type, and (for arrays) bounds of related %EXTERNAL and %EXTRINSIC declarations must be identical.

Running IMP programs

The command RUN is used to cause the execution of an IMP program. Before starting execution the program has to be loaded, as described in chapter 10. Normally all external references to %EXTERNAL routines and data entries are resolved before execution. At times however it is more efficient to delay linking until the first call is made on a routine function or map. This is useful when a run of a program is likely to use only some of the external routines called. In order to delay the loading of a particular routine it should be specified by a %DYNAMICROUTINESPEC instead of an %EXTERNALROUTINESPEC. Note that this technique cannot be used for %EXTERNAL data items. See notes on dynamic loading in chapter 10.

Unsatisfied References

If during loading a routine cannot be located to match an %EXTERNAL specification, then an appropriate message is printed but the execution of the program is allowed to continue up to the point where a call is made on that routine. At this point a message such as ILLEGAL CALL ON ROUTINE CHECKFILE is printed followed by an execution of %MONITORSTOP.

If an %EXTERNAL data entry is not found to satisfy an %EXTRINSIC reference then an area of store is allocated of the correct size, but it is not initialised. No failure message is printed.

LIBRARY ROUTINES

IMP System Library

Standard IMP library routines are available in the System Library, which is searched automatically following a search through any library indexes nominated by the user. The contents of this library are described in reference 9.

Graphics and other libraries

There are libraries of routines for accessing graphics devices and for other specialist purposes. In order to access routines in these libraries it is necessary to use the command APPENDLIB. This is explained in chapter 10.

Accessing EMAS foreground commands

It is possible for an IMP program to access directly any EMAS foreground command. As explained in chapter 18, the Subsystem comprises a large number of IMP routines and, in particular, there is one for each foreground command. All foreground command routines must be specified explicitly and all have one string parameter. For example, if a program is required to call the command DEFINE, it would have to include

```
%EXTERNALROUTINESPEC DEFINE(%STRING(63) S)
```

The string used in the call should contain the same text as would be typed within brackets when the command is typed at the interactive terminal. For example, if it is required to define STREAM23 to be a line printer then one would type the command

```
DEFINE(STREAM23,.LP)
```

Similarly, if the same command were called from within a program the routine call would be

```
DEFINE('STREAM23,.LP')
```

Notes

* Since within a program the parameter is a string, it must be enclosed in quotes or must be a string expression.

* When typing commands, spaces are removed from parameters. This is not done in the case of commands called from programs.

Chapter 18 contains a fuller description of the effect of calling commands from within programs.

Other IMP routines specific to EMAS

There are a number of external routines available to the IMP programmer which are specific to EMAS. When using them the programmer should be aware that his program may not be readily transferred to another operating system.

Interactive terminal handling routines

The routine PROMPT and the function TESTINT are used to exploit features of the interactive terminal.

PROMPT is used to set the text which is printed on the interactive terminal for subsequent input requests. It must be explicitly specified:

```
%EXTERNALROUTINESPEC PROMPT(%STRING(15) S)
```

While running user programs the default prompt is 'DATA:'. A call of PROMPT within the program will change this and the new text will be used as the prompt whenever input from the interactive terminal is required, until another call of PROMPT or a return to command level is made. No text is printed as a result of the call of PROMPT. It is the subsequent requests for input that result in the text being typed. If the user types ahead then the prompt is not printed at all. The following demonstrates the use of PROMPT:

```
%BEGIN
%EXTERNALROUTINESPEC PROMPT(%STRING(15) S)
%STRING(15) INFILE, OUTFILE
%INTEGER RECORD
    PROMPT('INPUT FILE:')
    READSTRING(INFILE)
    PROMPT('OUTPUT FILE:')
    READSTRING(OUTFILE)
    PROMPT('RECORDS:')
    READ(RECORD)
    .
    .
```

When run at the interactive terminal this might have the appearance:

```
COMMAND: RUN(CONVERT)
INPUT FILE: 'RES506'
OUTPUT FILE: 'OUTPUT 6'
RECORDS: 256
.
.
.
```

Notes

* The maximum length of a prompt is 15 characters.

* Any characters can be included in a prompt, including space and newline.

* If a null string is used no prompt is output.

The integer function TESTINT is used to enable a program to respond to interrupts typed at the interactive terminal. As explained in chapter 5, if the ESC key is pressed the result is a prompt 'INT:', to which the reply can be:

* CR (carriage return) - this causes the interrupt to be ignored.

* A single letter followed by CR - these single character interrupts are interpreted by the Subsystem (see chapter 6).

* A reply of 2-15 characters, terminated by CR - these interrupts are made available to user programs via TESTINT. Space characters in the reply are ignored.

TESTINT must be explicitly specified:

```
%EXTERNALINTEGERFNSPEC TESTINT(%INTEGER CONSOLE, %STRING(15) TEXT)
```

TESTINT takes two parameters: an interactive terminal number, which should currently be zero (but which may later be used to distinguished between different interactive terminals connected to a process), and a string of maximum length 15 characters. The latter is used thus:

* If it has the value null, i.e. a zero length string, then the effect is to clear any outstanding user interrupts. This should be used at the beginning of a program which tests for user interrupts.

  The result of the function in this case should be ignored; e.g.

  ```
  DUMMY=TESTINT(0,''); !CLEAR ANY INTERRUPT
  ```

* If the parameter is a string of between 2 and 15 characters the effect of the function will be to return 0 if no such interrupt has been typed, and some other value if an interrupt of the same text has been typed.

Note that the Subsystem holds a table of up to eight separate user interrupts. When TESTINT returns a non-zero result the appropriate entry in the table is deleted. If an interrupt is typed with the same text as one in the table it is ignored. Thus the order in which interrupts are given is not significant.

Example:

```
        DUMMY=TESTINT(0,''); !TO CLEAR INTERRUPTS
RESET:
        .
        .
        .
        %CYCLE; !MAIN PROGRAM LOOP
            %IF TESTINT(0,'RESET')#0 %THEN -> RESET
            %IF TESTINT(0,'STOP')#0 %THEN %EXIT
            .
            .
            .
        %REPEAT
```

Checking for the existence of a file

It is often necessary to know whether a file of a particular name exists. The function EXIST can be used. It must be explicitly specified:

```
      %EXTERNALINTEGERFNSPEC EXIST(%STRING(24) S)
```

The parameter should be a string or string expression containing the name of the file. The result is non-zero if the file exists (and is permitted to this user in at least one mode).

Example:

```
      %STRING(6) USER
      %STRING(8) FILE
      %EXTERNALINTEGERFNSPEC EXIST(%STRING(24) S)
          READSTRING(USER)
          READSTRING(FILE)
          %IF EXIST(USER.'.'.FILE)#0 %THEN %C
      PRINTSTRING('FILE EXISTS') %AND %RETURN
```

If testing for a file belonging to self the user name can be omitted:

```
      %IF EXIST('OUTPUT')#0 %THEN -> DESTROY OUTPUT
```

Obtaining information about the Subsystem

There is sometimes a requirement for a program to determine information about the environment in which it is running. The external integer function SSINFO returns the address of a record that contains certain information. At present the record has the following format. It may be extended at a later date, but this will not affect existing programs.

```
      %RECORDFORMAT RF(%STRING(8) USER, %BYTEINTEGER MODE, %C
          %STRING(8) START, %STRING(15) LOADLIB, MODLIB, %STRING(19) DELIVER)
```

The various elements contain information as follows:

USER	is the full 8-character job name; i.e. the first 6 characters are always the user's job number and in the case of programs running in background mode the 7th and 8th characters will be the particular job identifier.
MODE	=1 for foreground mode - i.e. with an interactive terminal connected. =2 for background mode.
START	is the log-on time for foreground sessions ('hh.mm.ss'); otherwise the contents are undefined.
LOADLIB	is the name of the first library index to be searched for entries and commands.
MODLIB	is the name of the library index modified by calls of INSERTFILE etc. Note that LOADLIB and MODLIB are normally the same; they will both be 'SS#LIB' unless another library index has been selected by a call of USERLIB.
DELIVER	is the current delivery information, set by the last call of the DELIVER command.

In the following program the user name and delivery information is printed out at the head of an output file:

```
      %BEGIN
      %RECORDFORMAT RF(%STRING(8) USER, %BYTEINTEGER MODE, %C
      %STRING(8) START, %STRING(15) LOADLIB, MODLIB, %STRING(19) DELIVER)
      %RECORDNAME INFO(RF)
      %EXTERNALINTEGERFNSPEC SSINFO
          .
          .
          .
```

```
        %ROUTINE PRINT HEADING
            INFO==RECORD(SSINFO)
            PRINTSTRING('USER:'.INFO_USER)
            NEWLINE
            PRINTSTRING('DEPARTMENT:'.INFO_DELIVER)
            NEWLINE
            %END
            .
            .
            .
```

Calling FORTRAN

The object files produced by the EMAS IMP and FORTRAN compilers are compatible, and subject to certain limitations imposed by the languages it is possible to make cross calls from one language to the other. This is described in reference 9.

IMP INPUT/OUTPUT

Four file access methods are provided for IMP:

* Streams - used for character input/output; use routines READSYMBOL, WRITE etc.
* Sequential binary file handling - SQFILEs; use routines READSQ, WRITESQ etc.
* Direct access file handling - DAFILEs; use routines READDA, WRITEDA etc.
* Store mapping - see chapter 13.

The routines and functions provided by the IMP language system are described in reference 1. The system-dependent aspects of these facilities are described below.

Linking logical channels to files and devices

The command DEFINE should be used to establish a link between a logical channel in a program and a particular file or output device. The first parameter for DEFINE sets the access method and the channel number. For IMP the relevant ddnames are:

Access Method	DDNAME	Abreviation allowed
Character I/O	STREAMn	STn
Sequential binary	SQFILEn	SQn
Direct access binary	DAFILEn	DAn
Store map	SMFILEn	SMn

Note that n should be a 1 or 2 digit number in the range 1-80. Examples:

```
DAFILE10
SQ17
ST7
SMFILE3
```

There must not be a conflict between the channel numbers used for different types of access. For example, a program cannot access stream 1 and sequential file 1. The other parameters for DEFINE are described in chapter 11.

CHARACTER I/O

Character I/O can be used both for input and output.

For input the following can be used:

* the interactive terminal (defined if necessary as '.TT')
* a character file (see chapter 11)
* a data file, so long as its contents are valid ISO characters (see chapter 11)

For output the following can be used:

* the interactive terminal (defined if necessary as '.TT')
* a character file. If the specified file does not exist it will be created automatically; if the file does exist it will be overwritten regardless of its type
* any of the following output devices, or their remote equivalents:

line printer	.LP
card punch	.CP
paper tape punch	.PP

Examples of valid DEFINE calls for IMP Streams:

```
DEFINE(ST22,.TT)
DEFINE(ST17,FILEIN)
DEFINE(ST80,.PP)
```

Default stream definitions

The following two tables show the definitions that are established by default, in foreground and background mode respectively:

Foreground Mode

Channel	Input/output	Device	Margins
0	Input	.TT	1:72
98	Input	.TT	1:72
0	Output	.TT	1:132
99	Output	.TT	1:132
95	Output	.PP	1:80
97	Output	.CP	1:80

Background Mode

Channel	Input/output	Device	Margins
0	Input	Job file	1:72
98	Input	Job file	1:72
0	Output	.LP	1:132
99	Output	.LP	1:132
95	Output	.PP	1:80
97	Output	.CP	1:80

Note that stream 0 is exceptional in that it can be used for input and output simultaneously. On entry to a program the input and output streams selected are both 0. Margins for other streams are, by default, as follows:

Types	Margins
Files	1:80
.TT	1:132
.LP	1:132
.CP	1:80
.PP	1:80

For user-defined streams with channel numbers in the range 1-80 the routine SET MARGINS can be used to change the margins for a given stream.

Size of stream files

The maximum size for a stream output file is determined from the current DEFINE for the channel on which it is being written. This applies to both files and output devices. The size is passed as the third parameter in DEFINE, and sets the size in Kbytes (1024 bytes). The default is 255 Kbytes, the maximum currently 1023 Kbytes.

Examples:

```
DEFINE(ST10,FILEA,500)
DEFINE(ST11,.LP,100)
```

Notes

* The size required is specified in Kbytes. It is rounded up to the next segment (64 Kbytes).
* For some output devices there is a lower limit - see chapter 4.
* See also chapter 11 for further information about DEFINE.

SEQUENTIAL BINARY FILES

The routines OPENSQ, CLOSESQ, READSQ, WRITESQ and READLSQ are available to access sequential binary files. The only type of file that can be used for sequential binary input is a data file (see chapter 11). For output, data files are written, regardless of the type of an existing file. Additionally the following output devices can be accessed:

Device	Abbreviation	Record Length
EBCDIC Card Punch	.ECP	80
Binary Card Punch	.BCP	160
Binary Paper Tape Punch	.BPP	80
Graph Plotter	.GP	80
Matrix Plotter	.MP	300

DIRECT ACCESS BINARY FILES

The routines OPENDA, CLOSEDA, WRITEDA and READDA are available for accessing direct access binary files. They can only be used in conjunction with data files having a fixed record length of 1024 bytes. When OPENDA is used on a channel which defines a non-existent file then a new file is created and filled with the unassigned pattern. The size of the file is extracted from the third parameter passed to DEFINE. For example, if it is required to create a file of 10 records, the following DEFINE could be used:

```
DEFINE(DA70,TESTDA,10)
```

Note that this command does not create the file TESTDA: it is also necessary to run a program that includes the statement

```
OPENDA(70); !CREATE AND OPEN FILE
```

See also the section on the command DEFINE in chapter 11.

STORE MAP FILES

Accessing files by mapping them onto variables and arrays in a program is described fully in chaoter 13.

EFFICIENCY OF IMP WHEN USED WITH EMAS

When writina IMP proorams snecifically for EMAS there are a number of ways in which greater efficiency can be achieved. The main aim should be to reduce the number of page turns. This is particularly important for programs run in foreground mode, since the number of page turns is an important factor in determining the elapsed time taken to run the program. The following points are suggested:

* Use a %CONST rather than an %OWN array where the contents of an array remain constant throughout a program.

* Do not use %OWN arrays just to achieve initialisation to zero. Instead use a normal array and a cycle to clear it to zero.

* Use arrays of the correct size. If the size varies significantly from one run to another use dynamic bounds.

* When accessing two-dimensional arrays remember that they are laid out in store in such a way that their first bound increases more rapidly; e.g.

  ```
  %INTEGERARRAY IN(1:200,1:200)
  ```

  is laid out thus:

  ```
  IN(1,1)
  IN(2,1)
  IN(3,1)
  .
  .
  .
  IN(200,1)
  IN(1,2)
  IN(2,2)
  etc.
  ```

  Where possible, when accessing a large array of this type, an attempt should be made to access the elements in the order in which they occur in store. For example, when clearing such an array to zero the following should be used:

  ```
  %INTEGER I,J
  %CYCLE I=1,1,200
      %CYCLE J=1,1,200
          IN(J,I)=0
      %REPEAT
  %REPEAT
  ```

These points should be seen in perspective. Most of them are only of importance in programs which access large data areas.

CHAPTER 16
FORTRAN ON EMAS

This chapter describes the environment in which a FORTRAN program runs on EMAS. EMAS provides a compiler for the FORTRAN IV language which is fully described in reference 2. The FORTRAN user, as against the IMP user, is at a disadvantage in exploiting some of the more sophisticated features of EMAS, but it is usually possible to overcome this by writing an interface routine in IMP (see later in this chapter).

Compilation

The command FORTE is used to compile a FORTRAN source file, as described in chapter 9. When preparing such a source file using an interactive terminal the TAB facility can be useful (see chapter 5). The command PARM is used to set compile time options. The file should contain one or more program units (subprograms), of which at most one should be a main program. If after successful compilation the object file is analysed using FILEANAL, it will be found to contain the following:

* a MAIN PROGRAM entry, if the file contained a main program
* an entry for each SUBROUTINE, FUNCTION or ENTRY statement
* a data entry for each named COMMON statement in a BLOCK DATA subprogram

Subroutine linking

CALL statements, and references to functions declared as EXTERNAL, result in cross references being included in the load data part of the object file. It is important to appreciate that the only information included in the reference is the name - in fact only the first 6 characters of the name. There is no information held to indicate what type of subprogram is being referenced. Thus great care should be taken to specify the correct type and parameter list in subroutine and function calls. Calling the wrong type of subprogram, or using the wrong number or types of parameters, can result in faults that are very difficult to diagnose.

Data linking

The COMMON statement is used to provide access to data used in more than one subprogram. Blank COMMON blocks in different subprograms are automatically equivalenced when a program is loaded. Named COMMON blocks are equated to all other COMMON blocks with the same name, and are initialised if a BLOCK DATA subprogram is included which refers to the same named COMMON block.

Running FORTRAN programs

The RUN command is used to cause execution of a FORTRAN program. The parameter should be the name of an object file which includes a main program among its subprograms. Before starting execution the object file must be loaded, as described in chapter 10. Normally all references to subroutines and functions are satisfied before starting execution, appropriate messages being printed for any that are not found. If some are not found execution is nonetheless allowed to proceed, up to the first call of a routine which was not found. At this point a message is printed, such as:

```
        ILLEGAL CALL ON ROUTINE MAPTOP
```

and then diagnostics are printed followed by a STOP.

Dynamic loading

As explained in chapter 10 there are some situations where it is expedient to delay subprogram loading until the first call on a subprogram. This is particularly useful in programs which during any one run only call on a small number of the subprograms that they reference in total. This dynamic linking is only relevant to programs that are divided among several object files. The suggested strategy is as follows:

* Create one file which contains the main program and the subprograms which are always used in every run.

* Compile this without using the compile time option DYNAMIC, together with any other required options.

* Divide the rest of the subprograms into functional groups such that, as far as possible, any one file contains only references to subprograms within itself or in the main file.

* Compile these files without the DYNAMIC option selected.

It is possible to make more divisions, but as explained in chapter 10 there are disadvantages in taking this technique too far. The following points should also be noted:

* A file that is compiled with the option DYNAMIC is not, itself, loaded dynamically - it is the object files that it accesses that are loaded dynamically.

* The blank COMMON area defined in the main program file should be at least as large as that used by any of the dynamically loaded subprograms.

* All BLOCK DATA subprograms should be included in the main program file, unless all references to any one of them are confined to one of the subsiduary files, in which case it can be included in that file.

LIBRARY ROUTINES

System library

A FORTRAN system library is automatically searched for the intrinsic and mathematical function subprograms. Details of these subprograms are contained in reference 9, and a summary table is included in reference 2.

Graphics and other libraries

There are libraries of graphics routines and other specialist routines available on EMAS (reference 8). Additionally the Numerical Algorithms Group library is available to the FORTRAN programmer. Further information should be obtained from the Advisory Service.

Accessing EMAS foreground commands

EMAS foreground commands can be accessed from within FORTRAN programs. Chapter 18 discusses some of the wider implications of doing this, but the mechanism is described here. EMAS foreground commands are in fact IMP external routines which are normally called as a result of the Subsystem interpreting the command typed on the interactive terminal. There are two pieces of information involved:

* The name of the command - only the first 8 characters are significant.

* The parameter passed - this is the text enclosed in brackets after the command name (with spaces and newlines removed), it is passed as a %STRING parameter.

The main problem of calling these routines from Fortran is that the language does not include string variables. However an intermediate routine EMASFC is available to convert literal constants into strings. EMASFC takes four parameters.

Example:

```
      CALL EMASFC('DEFINE',6,'FT10,.LP',8)
```

The first parameter is a literal constant which contains the name of the command being called, and the second is its length. The third parameter is another literal constant which contains the parameter to be passed to the command and the fourth contains the length of this constant. The example above would have the same effect as typing

```
      DEFINE(FT10,.LP)
```

at COMMAND level.

Note that if a command is used which requires no parameter, then a dummy parameter, with a given length of 0, must still be included in the call of EMASFC.

Example:

```
      CALL EMASFC('METER',5,'DUMMY',0)
```

Changing the PROMPT text

There is a FORTRAN routine FPRMPT which is equivalent to the IMP PROMPT routine, and is used to set the text of the message used to prompt for input from the interactive terminal. Note that no output is generated directly as a result of a call on FPRMPT - the text is only printed at the time of the next request for input, and then only if the user has not typed ahead. The same prompt is used until the next call of FPRMPT or return is made to command level. The routine takes two parameters: a literal string of not more than 15 characters, and an integer to indicate the length of the string:

```
      CALL FPRMPT('RUN NUMBER:',11)
```

In the following example FPRMPT is used in conjunction with WRITE and READ statements:

```
          WRITE(6,100)
   100    FORMAT(' NOW TYPE IN CONTROL VALUE')
          CALL FPRMPT('NO. OF VALUES:',I4)
          READ(5,101)NOV
   101    FORMAT(I2)
          CALL FPRMPT('VALUE:',6)
          DO 1 I=1,NOV
     1    READ(5,101)VALUE(I)
          .
          .
          .
```

The resulting dialogue on the interactive terminal might have the following appearance:

```
      NOW TYPE IN CONTROL VALUES
      NO. OF VALUES: 3
      VALUE:27
      VALUE:10
      VALUE:99
```

Note that Edinburgh Fortran includes a free-format READ facility which simplifies the operation of reading data from the interactive terminal. This is fully described in reference 2.

Calling IMP

The object files produced by the FORTRAN and IMP compilers are compatible, and within the limits imposed by the parameter passing facilities of the two languages it is possible to make calls between object files of programs written in different languages. This is particularly useful to the FORTRAN programmer in that it makes it possible to exploit some of the more sophisticated facilities of EMAS. The mechanisms for making calls from FORTRAN to IMP are described in reference 9.

Access Methods

EMAS provides the necessary support routines for the input/output facilities of the FORTRAN language. There are two main access methods: sequential and direct access, and both allow for the optional use of a FORMAT statement.

File types

The following table indicates which file types and devices can be used for each access method. The file types are described in chapter 11, the output devices in chapter 4.

	Sequential		Direct Access	
	With FORMAT	Without FORMAT	With FORMAT	Without FORMAT
Input	Terminal (.TT) Character File Data File	Data File	Data File	Data File
Output	Terminal (.TT) Line Printer (.LP) Card Punch (.CP) Paper Tape Punch (.PP) Data File Character File	Binary Card Punch (.BCP) Binary Paper Tape Punch (.BPP) Data File	Data File	Data File

Use of DEFINE with FORTRAN

The command DEFINE is used to establish a link between a logical channel number in a FORTRAN program and a particular file or device. The command, and the associated commands CLEAR and DDLIST, are described fully in chapter 11. When used with FORTRAN the normal form of the command should be

```
      DEFINE(FTn,filename)
   or DEFINE(FTn,device)
```

For example:

```
      DEFINE(FT23,DATA107)
      DEFINE(FT1,.LP)
```

Sequential input

It is possible to use the interactive terminal (.TT), a character file or a data file for input (see the table above). When using the interactive terminal or a character file the record is padded with blank characters up to column 80 if necessary. This facilitates the transfer of programs from card-oriented systems. For example, if it is required to read a title of up to 80 characters from the interactive terminal, the user has only to type the printable characters - the rest will automatically be filled with spaces.

Example:

```
        INTEGER*4 TITLE(20)
   100  FORMAT(20A4)
   101  FORMAT(' ',20A4)
        CALL FPRMPT('TITLE:',6)
        READ(5,100)TITLE
        WRITE(6,101)TITLE
```

When run this would appear on the interactive terminal (assuming default channel definitions) as:

```
TITLE:FIRST EXPERIMENT
FIRST EXPERIMENT
```

Note that when reading from a data file, on the other hand, the length of the input record must be at least as long as that implied by the FORMAT being used. Otherwise the FORTRAN run time fault

```
RECORD WRONG LENGTH
```

will occur.

Sequential output

Sequential output written using FORMAT control can be directed to data files, character files, the interactive terminal, or various character output devices (see the table above). It is important to understand the effect of the print control character which is often put at the start of the output record. If the output is directed to a device that can interpret this - for example, the interactive terminal or a line printer - then it is interpreted as explained in reference 2. If, instead, it is directed to a file or device such as a card punch, then it is treated as part of the output record. One of the effects of this is that if output intended for eventual printing is first put in a file and then listed using the LIST or SEND command, the print control character is printed as the first character of the line and is not used to control the line spacing of the printer. This problem can be overcome by specifying the record format of the file to be VA or FA. This is fully described in chapter 11, under DEFINE.

The following example shows this:

```
        .
        .
        .
   100  FORMAT('1 HEADING OF OUTPUT')
        WRITE(8,100)
   101  FORMAT('0',3I10)
        DO 1 I=1,30
     1  WRITE(8,101) RES(I), TOP(I), SEN(I)
        .
        .
        .
```

If the command

```
DEFINE(FT8,.LP)
```

is typed before running this program the output will be printed on the line printer, correctly interpreting the print control characters for 'newpage' and 'two new lines'. If however it is required to put the output in a file and then to print it, the file could be defined thus:

```
DEFINE(FT8,OUTLP,,VA133)
```

The effect of the A in the record format (VA) is to mark the file in such a way that when it is listed on an appropriate device the first character of each record is treated as a print control character. If, instead, the command

```
DEFINE(FT8,OUTLP)
```

were used before running the program, and later the command

```
LIST(OUTLP,.LP)
```

were used, the listing would contain the print control characters '1' and '0' at the beginning of the lines of output, and the results would be printed on consecutive lines with no newpage at the start.

When writing output for subsequent rereading by another FORTRAN program, there is no need to include a print control character in the FORMAT. However, in this situation it is more efficient not to use FORMAT control at all.

Direct access files

When using direct access files the following points should be noted:

* Only files can be used, not devices.
* The size and record length of the file are not taken from the DEFINE command: they are determined from the DEFINE FILE statement in the FORTRAN program, which is used when the file is first referenced. The size parameter in the DEFINE command used when the file is created must be at least as large as that required for the file; the default is 255K.

Default definitions

When running in foreground mode the following definitions are established by default. They can be overridden by use of the DEFINE command.

Channel	Input/Output	Device
5	Input	Interactive terminal (.TT)
6	Output	Interactive terminal (.TT)
7	Output	Card punch

When running in background mode the following default definitions are established; they too can be overridden by the use of DEFINE.

Channel	Input/Output	Device
5	Input	Job file
6	Output	Line printer
7	Output	Card punch

Note that the effect of defining channel 5 as the Job file is that the data to be read on channel 5 can follow immediately after the RUN command in the Job file (see chapter 19). For example, the following program could be run using the control file shown:

```
        Program                            Control file

        .                                  RUN(FORT73)
        .                                   276 389
        .
 100    FORMAT(2I4)
        READ(5,100)NUM,TOP
        .
        .
        .
```

Closing FORTRAN files

The FORTRAN language does not include explicit OPEN and CLOSE statements for files. Instead they are opened automatically when they are first accessed and closed automatically by the Subsystem on return to command level. There are a few occasions when it is useful to be able to close a file explicitly from FORTRAN. The EMAS routine CLOSEF is available. Its parameter should be an INTEGER*4 constant or variable containing the logical channel number of the file to be closed. Example:

```
      CALL CLOSEF(27)
```

CHAPTER 17
ALGOL ON EMAS

This chapter describes the way in which an ALGOL program can be compiled and run on EMAS. The ALGOL compiler is described in reference 3. Note that examples of ALGOL in this chapter use the same graphical representation of the language as that used in the ERCC ALGOL Language Manual.

Compilation

An ALGOL source file can be compiled using the command ALGOL, as described in chapter 9. The related command PARM is used to set compile time options.

Contents of source file

A source file can consist of a begin - end block (possibly including blocks and procedures), in which case it is regarded as a complete program and is executed by use of the RUN command. Alternatively it can consist of one or more complete procedures not contained in a begin - end block. These are not intended to be executed independently but are for calling from other ALGOL programs or procedures. This topic is covered fully in reference 3.

Running an ALGOL program

Before execution, an ALGOL program is loaded as described in chapter 10. Any references to separately compiled procedures that cannot be satisfied are listed and execution commences. If a call is made on an unsatisfied reference the run terminates, with a failure message of the form

```
ILLEGAL CALL ON ROUTINE name
```

followed by diagnostic information.

Library routines

Reference 3 describes the procedures available in the system library for the ALGOL user. Information about other procedures can be obtained from the Advisory Service.

Accessing EMAS foreground commands

Currently there is no facility for allowing an ALGOL program to make a call on an EMAS command. The restriction is imposed by the fact that ALGOL string parameters can only be passed by name. The simplest way to overcome this problem is to write an interface routine in IMP. In the following example the IMP external routine ADEFINE takes one string name parameter. It makes a call on DEFINE using the same parameter.

```
%EXTERNALROUTINE ADEFINE(%STRINGNAME AS)
%EXTERNALROUTINESPEC DEFINE(%STRING(63) S)
 DEFINE(AS); !CALL ON DEFINE USING PARAMETERS PASSED (FROM ALGOL) IN AS
%END
%ENDOFFILE
```

The ALGOL program should include a specification of the procedure ADEFINE and a call, as in the example below:

```
begin
procedure ADEFINE(S);
string(S);
external;
    .
    .
    .
ADEFINE([ST3,.LP])
    .
    .
    .
end
```

Chapter 18 of this manual includes further information about calling EMAS commands.

Calling other IMP and FORTRAN routines

Reference 3 describes the mechanism by which ALGOL programs can access IMP and FORTRAN.

ALGOL INPUT/OUTPUT

EMAS ALGOL includes a comprehensive set of input and output procedures for accessing character files. The DEFINE command should be used to establish a link between a logical channel and a file or output device (see chapter 11). The only valid access method is STREAM; for example:

```
DEFINE(ST27,DALG27)
```

The procedures share the stream handling facilities of IMP, full details of which are given in chapter 15 under the heading CHARACTER I/O. This includes a table of default stream definitions. Currently there are no binary file handling procedures provided for ALGOL. It would be possible for an ALGOL program to access IMP sequential or direct access file-handling facilities via interface routines, in a similar way to the example above of accessing DEFINE.

CHAPTER 18
CALLING COMMANDS FROM WITHIN PROGRAMS AND WRITING COMMANDS

A fundamental design intention of the standard Subsystem is that all the facilities and functions available as commands should also be available from within programs. The converse of this is that suitably written programs can be used as commands. This chapter is written primarily for the IMP user. Chapters 16 and 17 explain how EMAS commands can be accessed from FORTRAN and ALGOL.

The command interpreter

The command interpreter is that part of the Subsystem that reads the text typed in response to a COMMAND: prompt. It converts it into two parts:

* A string of up to 8 characters, which constitutes the name of the command being called (truncated if necessary)

* A string of up to 63 characters (from which any spaces and newlines are removed), which is used as the parameter string for the command

Next a routine of the same name as the command is loaded (see chapter 10) and called with the parameter provided passed to it. Thus for each command in the Subsystem there is a routine with the following specification:

```
%EXTERNALROUTINESPEC command(%STRING(63) PARM)
```

Calling commands from within programs

The method of calling a command from within a program is as follows:

* Specify the routine with a %SPEC statement of the form given above; for example:

  ```
  %EXTERNALROUTINESPEC DEFINE(%STRING(63) S)
  ```

* Call the routine, passing as a parameter a string expression containing text, exactly as would be passed to the command at the interactive terminal. Note however that spaces and newlines will not automatically be removed - hence they should not be included.

In the following example a program is shown which reads a file name from the interactive terminal and then calls DEFINE to link the file to stream 1.

```
%EXTERNALROUTINESPEC DEFINE(%STRING(63) S)
%STRING(8) FILE
      .
      .
      .
      READSTRING(FILE)
      DEFINE('STREAM1,'.FILE)
      .
      .
      .
```

Note that some commands do not require a parameter, for example METER. In this case the command should still be specified with a parameter and called with a null string as its actual parameter:

```
%EXTERNALROUTINESPEC METER(%STRING(63) S)
    .
    .
    .
    METER('')
```

If this procedure is followed, programs will continue to work even if optional parameters are provided for these commands at a later date.

Commands which read input or generate output

Many commands generate output and some, for example CONCAT, require input apart from their parameters. The standard Subsystem commands operate according to the following rules:

* Input is normally read from the currently selected input stream. For example, if the sequence

  ```
  SELECTINPUT(3)
  CONCAT('')
  ```

  were obeyed, then CONCAT would read its control data from the file defined as stream 3.

* The exceptions to this occur when the input to the command is contained in a file specified as a parameter. In this case the input is read from the specified file and then control is returned to the calling program with the current input channel selected. The command routine itself contains code to do the following:

  * note the user's current input stream
  * define and select the specified input file
  * select back to the user's current input stream

  One side effect of this is that if the call is made when the program has read only part of the current line then the rest of that line is lost (see SELECTINPUT, in reference 1).

* Output is normally sent to the currently selected output stream.

* If an explicit output file is nominated in the command then it is used and the user's output stream is re-selected before returning to his program. This can have the effect of putting an extra newline character in the user's output (see SELECTOUTPUT, in reference 1).

File protection mechanism

The Subsystem includes checks to prevent the user from carrying out operations on files which are incompatible with their current use. For example, in the following program an attempt is made to destroy a file which is currently selected for output.

```
DEFINE('ST80,OUT')
SELECTOUTPUT(80)
    .
    .
    .
DESTROY('OUT')
```

In this situation the operation would fail with the message

```
DESTROY FAILS - FILE OUT CURRENTLY IN USE
```

The problem can be overcome by closing the file, which in this program could be achieved by inserting two statements before the DESTROY:

```
SELECTOUTPUT(0);    !OUTPUT TO CONSOLE
CLOSESTREAM(80)
```

The same failure message will occur if any of the following types of activity are attempted in respect of a currently open data file:

* DISCONNECT
* RENAME
* OFFER
* Any command that would write to the file; for example, FLIST('OUT')

Apart from files which are currently in use for Input/Output operations, files which are currently loaded (see chapter 10) are also protected from corruption. For example, if the object file RPROGY contained the statement

```
IMP('RPROGS,RPROGY,.LP')
```

then this call of IMP would fail because an attempt was being made to write to a currently loaded file. The examples given earlier of invalid use of files for Input/Output also apply to loaded files. If an attempt were made to destroy a currently loaded file the message would be:

```
DESTROY FAILS - FILE name CURRENTLY BEING EXECUTED
```

Protection of file definitions

File definitions set up by DEFINE or DEFINEMT are protected if the channel they reference is currently open. For example, the second DEFINE in the example below

```
DEFINE('ST3,FIRST')
SELECTINPUT(3)
    .
    .
    .
DEFINE('ST3,SECOND')
    .
    .
    .
```

would fail with the message

```
DEFINE FAILS - FILE CURRENTLY DEFINED AS STREAM03 IS OPEN
```

Again, use of CLOSESTREAM will solve this problem.

Special precautions with OBEYFILE

OBEYFILE can be called from within a program. It should be noted, however, that all files left open at the time of the call on OBEYFILE will be closed after it has completed - as is normally the case on return to command level.

Use of RUN

RUN can be called from within a program or an %EXTERNAL routine. It can be nested, i.e. the called program can also call RUN, and a new environment will be established for each call. This means that a fault which occurs in an inner program will return control to the RUN which called it and so on back to the Subsystem command level. Note however that only the first 6 levels of calling can have separate environments; thereafter they will use the environment of the sixth program. On returning from a program which has been RUN from another program, files are not automatically closed. They should be closed explicitly if necessary. It is possible, though perhaps not very useful, for a program to call itself recursively:

```
      %EXTERNALROUTINESPEC RUN(%STRING(63) S)
      %BEGIN
          .

          .
          .
          %IF .... %THEN RUN(RECUY); !RECUY IS THE OBJECT FILE OF THIS PROGRAM
          .
          .
          .
      %ENDOFPROGRAM
```

Detecting errors in Subsystem commands

When Subsystem commands typed on the interactive terminal fail they output appropriate error messages on the interactive terminal. When called from within a program they output the same error messages on the currently selected output stream. These messages can be suppressed by making a call on the external routine SSFOFF (see example below). By interrogating the function SSFAIL a user program can determine whether a particular command has worked correctly. A non-zero result from this function indicates that the command has not worked correctly. Further, in the case of a failure the string function SSFMESSAGE will return a message indicating the nature of the fault. The following example shows the use of these three facilities:

```
      %EXTERNALROUTINESPEC DEFINE(%STRING(63) S)
      %EXTERNALROUTINESPEC SSFOFF
      %EXTERNALINTEGERFNSPEC SSFAIL
      %EXTERNALSTRINGFNSPEC SSFMESSAGE
      %EXTERNALROUTINESPEC PROMPT(%STRING(15) S
          SSFOFF; !SUPPRESS STANDARD FAILURE MESSAGES
          .
          .
          .
          PROMPT('OUTPUT:')
GOUT:     READSTRING(OUT)
          DEFINE('ST80,'.OUT)
          %IF SSFAIL#0 %START; !SOME FAILURE IN DEFINE
             PRINTSTRING(SSFMESSAGE.' TRY AGAIN.')
             -> GOUT
             NEWLINE
          %FINISH
```

Notes

* SSFOFF remains in effect until the return to command level

* SSFAIL and SSFMESSAGE should be interrogated immediately after the command which is to be checked has been called

Writing own commands

It is a straightforward matter to write one's own commands. The structure of IMP external files is described in chapter 15 and in reference 1. Commands can, but need not, call Subsystem commands or other user written commands. %EXTERNAL routines written to be used as commands must have one parameter, of type %STRING(63).

When such a command is called, its parameter will contain all the characters contained in the parentheses when the command was typed, with the spaces and newlines removed.

In the following example a command has been written to simplify the calling of the IMP compiler. The user has decided to adopt the convention that for a given program his source and object files will have the same name except that they will have S and Y respectively as their last characters. Hence to compile PROG27S into PROG27Y he will type:

```
      I(PROG27)
```

The routine required is as follows:

```
%EXTERNALROUTINE I(%STRING(63) FILE)
%EXTERNALROUTINESPEC IMP(%STRING(63) S)
!CALL IMP COMPILER WITH NAMES FILE.'S' AND FILE.'Y'
    %UNLESS 0<LENGTH(FILE)<=7 %START; !CHECK LENGTH OF NAME
        PRINTSTRING('INVALID PARAMETER TO I')
        NEWLINE
        %RETURN
    %FINISH
    IMP(FILE.'S,'.FILE.'Y,,.TT'); !COMPILE WITH FAULT OUTPUT TO .TT
%END; !OF ROUTINE I
```

Conclusion

The facilities described above, together with those described in chapter 15 (for example PROMPT), make it possible to build a specialist interface with an appearance that is more meaningful to the end user. This is particularly useful in situations where the end user is not conversant with computers. It is possible to use command names that have meanings to him or her, and to provide arrangements to check replies to requests for input. Further, these facilities make it possible for the experienced user to simplify his own actions in using the Subsystem, by changing its appearance to suit his requirements.

CHAPTER 19
RUNNING WORK IN BACKGROUND MODE

This chapter is concerned with running jobs in background mode (see chapter 6). Two separate methods can be used to define such jobs:

* They can be defined in a job control language based on the IBM 360 job control language. This facility was provided initially to simplify the transfer of work from the ERCC 360/50 to EMAS. The future of this method is uncertain and users are advised to use the second method.

* They can be defined in terms of the command language used for foreground sessions. This method of initiating jobs is described below.

Additionally the chapter describes the facility provided for sending jobs to remote processors.

Preparing the job file

The job file can be read in on cards or paper tape or created using the Editor. It should contain a sequence of commands and data as might be typed at a foreground session. For example, the following file could be used to compile a program and run it with some data read from stream 2.

```
PARM(NOARRAY,NOCHECK)
IMP(TESTB,TESTBY,.LP)
DEFINE(ST2,.TT)
RUN(TESTBY)
'TEST RUN' 4 7 8 8 -99
```

Notes

* The format of the file is identical to that used for the command OBEYFILE (see chapter 21).

* The user does not need to provide any job identification information. This is added by the Subsystem.

* It is not necessary to terminate the job with the command STOP as one would terminate a foreground session. The job is terminated automatically when end-of-file is reached.

* In order to read data from the file itself the relevant stream is defined to be the 'terminal' (see the DEFINE in the example above), since the file is a direct replacement for the user typing at his terminal.

Running the job

The job can be put into the background job queue by use of the command DETACH. It will be run after the user has logged off according to scheduling rules defined below. DETACH takes four parameters:

1. The filename of the job file: this is the file prepared as above. Note that a copy of the file is made by the Subsystem, so that subsequently altering or destroying the file will not affect the job once it has been DETACHed. More than one file can be concatenated; for example

```
DETACH(START+DATA+TIDY)
```

2. Keywords:

   NOW means that the user is willing to have the job run at any time after he logs off.

   LOW means that the job should be run with low priority but charged for at a reduced rate (see Job Scheduling, below).

   If this parameter is omitted the job will be run at standard priority (see below). This parameter can be used instead to indicate the destination of a job being DETACHED to another computer (see 'Detaching jobs to other computers', below).

3. CPU Time Limit for the whole job, in minutes. This information is used for scheduling purposes (see below), and to impose a limit on the time used for the job. The default is 2 minutes, the maximum 120. Note that if any of the individual commands in the job is likely to take more than the default command limit (2 minutes) it will be necessary to include a call of the command CPULIMIT (see chapter 21) in the job file, as well as giving the estimated total job time as this parameter.

4. This parameter can be used to name a file or output device for the job. This is by default the local line printer. The file or device specified will contain a listing of all the commands and all the output that would, in foreground mode, appear on the terminal. If the file does not exist one will be created and if it does it will be overwritten. This facility is useful for obtaining the listing on a remote printer or for storing it in a file for subsequent access, particularly for users who cannot conveniently obtain their output from the local line printer.

   Example:

   ```
   DETACH(JOBFILE,NOW,3)
   DETACH(NN102,,20,.LP14)
   DETACH(ABJOB,LOW,60,ABOUT)
   ```

If the job is detached successfully a confirmatory message is typed which also informs the user what name has been given to the job. This is the name that should be used if referring to the job with the commands FINDJOB or DELETEJOB (see below).

Failures using DETACH

See Appendix 1: Subsystem Error Messages. In addition, failures will occur when DETACH is invoked if:

* An invalid keyword is used for the second parameter.

* A time limit outwith the range 1-120 is used.

* The user's file index is nearly full. This is a precaution to prevent a user detaching a job which will almost certainly fail because the file index is full. The cure usually involves destroying unwanted files.

Job scheduling

The precise meanings of the priority parameters are subject to change. At the time of writing the schedule for running background jobs is as follows:

0900 - 2100	NOW jobs only, in an order determined by their time limit and order of arrival. The number of NOW jobs run during the daytime depends on the load of foreground use.
2100 - 0900	NOW jobs and standard jobs, with no distinction between them, in an order determined by their time limit and order of arrival. When all of these are completed, LOW priority jobs, in an order based on time limit and order of arrival.

For charging arrangements for LOW priority jobs, see chapter 20.

The command FINDJOB

This command is used to obtain information about a job that has been put into the background job queue by a call of DETACH.

The command can be used in two ways:

* to obtain information about a specific job. In this case the name of the job is given as a parameter;.for example

```
FINDJOB(ERCC0603)
```

* to obtain information about all jobs in the batch queue for this user, in which case no parameter is used:

```
FINDJOB
```

The resulting messages include the name of each job, the time limit and the name of the command file(s) DETACHed to make each job.

The command DELETEJOB

This command is used to delete a job that has previously been put into the background job queue by a call of DETACH. The parameter should be one jobname, or a list of jobnames.

Example:

```
DELETEJOB(ERCC0600,ERCC0607)
DELETEJOB(ERCC0601)
```

Detaching jobs to other computers

An alternative use of DETACH is for sending jobs to other computers. Currently, for most users, it can only be used for sending jobs to NUMAC. The file (or concatenated files) being sent should contain one job, including any job control statements, program source or data if required, in exactly the form that would be used if the job were submitted on cards. The details of the services available at NUMAC are described in reference 10. The second parameter to DETACH is used to nominate the processor to which the job is to be sent. Currently NUMAC is described by the abbreviation .P37, though this may change. Hence:

```
DETACH(JOB370,.P37)
DETACH(JCL+PROG+DATA23,.P37)
```

Note that, as with detaching jobs to the local job queue, a copy is made of the file to be sent, so the original files can be reused or destroyed as soon as the DETACH command has been accepted. The command QUEUES (chapter 21) can be used to determine whether the job is still waiting to be dispatched to NUMAC.

CHAPTER 20
ACCOUNTS AND USAGE INFORMATION

This chapter describes the method of charging users for their use of EMAS resources, the procedure to follow to obtain access to the system, and the related command PASSWORD. Finally there is a description of two information commands METER and USER.

GAINING ACCESS TO THE SYSTEM

Before using EMAS it is necessary to obtain authorisation. This is in two parts:

* Obtaining authority to use the services of the Edinburgh Regional Computing Centre. This is dealt with by the ERCC User Support Group.

* Obtaining accredited EMAS user status. This is dealt with by the EMAS Operations Manager.

The end product is a 6-character user name and two four-character passwords. The first password is used for logging into an interactive terminal for foreground access to the system; the second is used for card and paper tape input (see chapter 4). Either or both passwords can be changed by use of the PASSWORD command.

The command PASSWORD

This command can be used to change either or both passwords. Passwords should consist of any four printable characters other than comma. The command take two parameters, the foreground and background passwords respectively:

```
PASSWORD(FORE,BACK)
```

If it is only required to change one of them then the other can be omitted:

```
PASSWORD(7777)
PASSWORD(,CARD)
```

CHARGING FOR USE OF RESOURCES

Resources are charged for, and invoices are sent to the appropriate funding body. The User Support Group can provide further details of the accounting mechanism outwith EMAS itself. There are two types of charge:

* charges for file space

* charges for computing resources used

File space charging

There are three rates for charging for file space. Files held in the immediate store (on disc) are charged at two rates. The charge for files which are CHERISHed is higher than for the rest. This is in order to recover the cost of backing up the files and replacing them on disc in the event of a serious system or hardware failure. Files held in the archive store (on magnetic tape) are charged for at a lower rate. This reflects the lower cost of keeping material on magnetic tape rather than on the disc file, and the fact that the archive store is more easily extended than the immediate store. The current (August 1976) charges are:

Type	Charge (pence) per page per day
Immediate Store (CHERISHed)	0.23K
Immediate Store (un-CHERISHed)	0.14K
Archive Store	0.014K

where K is a constant. Currently it is 3.45 for most users (6.90 for commercial users).

Charges for computing

There are four elements to the charge made for computing:

* Central processor time (T) - this is the time in seconds during which a user's process is actually executing instructions, plus an allowance to represent the share his process makes of facilities provided by the resident supervisor.

* Page turns (pt) - this is a count of the number of pages brought into main store or written back to the disc or drum for a user process. See Chapter 1 for a description of paging.

* Connect time (ct) - this is the time, in seconds, during which an interactive terminal is connected to the process.

* I/O unit records (U) - this is a count of the number of unit records handled; for example, lines printed or cards punched.

The other elements in the calculation are

* Priority (P) - this is normally 1 but is reduced to 1/3 for jobs that are detached at 'LOW' priority (see chapter 19).

* A constant K - this depends on the class of user. Currently (August 1976) this is 3.45 for most users (6.90 for commercial users).

The charge is calculated using the following formula:

$$\text{charge (pence)} = K\left[P\left(T + \frac{pt}{250} + \frac{ct}{50}\right) + \frac{U}{300}\right]$$

Project codes

It is possible for one user to divide his computing charges among various projects. This is done by using the command PROJECT, which accepts as its parameter a two-character project code. The characters should be upper case letters or digits. For example:

```
PROJECT(A9)
```

All work done after typing this command will be charged to code A9 until either PROJECT is used again, or the user logs off. The User Support Group should be consulted about the way in which project codes are printed in user accounts.

The command METER

This command is used to obtain usage information and an indication of the amount charged thus far in the current session (i.e. since log-on). The command takes no parameter. Output is of this form:

```
        21/09/76  12.44.23  CPU= 6.15 SECS  CT=21 MINS  PT=4282  CH=168P
```

The information given is as follows:

* Current date and time
* CPU time, in seconds
* Interactive terminal connect time, in minutes
* Page Turns
* Approximate charge (on the assumption that this is a user charged at the standard rate). No allowance is made in this charge for unit record output.

The command USERS

This command, which takes no parameter, is used to print out the number of active processes on the user's System 4. This normally includes three system processes (see chapter 2). It provides an indication of the loading on the system and the response that can be expected. Currently each machine can run with up to about 45 users.

CHAPTER 21
ANCILLARY COMMANDS

The table below lists the ancillary commands available in the Subsystem - commands which do not readily fit into any of the categories covered by earlier chapters, or, as in the case of OPTION for example, commands which relate to several of the categories.

Command	Purpose
CPULIMIT	Used to set time limit for subsequent commands
DELIVER	Used to specify delivery information, to be printed on output files
OBEYFILE	Used to execute a sequence of commands
OPTION	Used to set a number of optional characteristics of the Subsystem
QUEUES	Used to print information about files awaiting output
STOP	Used to terminate foreground session
SUGGESTION	Used to send suggestion to the System Manager

The command CPULIMIT

This command is used to set the amount of central processor unit (CPU) time allowed for subsequent commands. It takes one obligatory parameter, the time in minutes, and optionally a second parameter to specify time in seconds. For example

Command	Time set
CPULIMIT (3)	3 minutes
CPULIMIT (,10)	10 seconds
CPULIMIT (1,30)	1 minute and 30 seconds

Notes

* In order to provide good response for users of interactive programs there is a low limit imposed during busy periods on the maximum CPU time that can be set using CPULIMIT.

* Currently the maximum values for foreground use varies between 30 seconds and 10 minutes depending on the number of users logged on. For background jobs it is 120 minutes.

* The default setting for foreground access is 30 seconds and for background jobs it is 2 minutes.

* The command takes effect for the following and subsequent commands, and remains in effect until CPULIMIT is used again or the user logs off.

* This command can be useful for testing programs that contain faults which result in infinite loops. By using CPULIMIT with a low value - perhaps 5 seconds - the elapsed time taken to reach the TIME EXCEEDED failure is considerably reduced.

* CPULIMIT has no effect on the OBEYFILE command (see below) but affects any commands called by OBEYFILE.

The command DELIVER

This command is used to specify the text to be printed at the start and finish of output files to assist Job Reception staff in distributing output. The parameter should be suitable text with a maximum length of 19 characters. No spaces should be used. The underline character is a suitable substitute.

Example:

```
DELIVER(ALISON.HOUSE)
DELIVER(CHEMISTRY_K.B.)
```

Notes

* The registered name of the owner of the process is always printed on the output as well as the delivery information.

* The command takes effect immediately and remains in effect until another use of the DELIVER command.

* The form DELIVER(?) can be used to determine the current delivery information. It is printed in reply on the user's interactive terminal.

The command OBEYFILE

This command is used to execute a sequence of commands. The required commands and any data they would normally read from the interactive terminal should be put in a file, using the Editor or some other means. The format should be identical to that which would be used when typing commands on the interactive terminal. OBEYFILE takes one obligatory parameter, the name of the file containing the commands to be obeyed. Additionally a second parameter can be given to specify a file or device to be used for output. By default the output goes to the interactive terminal. For example:

```
OBEYFILE(NE26)
OBEYFILE(NRJOB,.LP14)
```

Among the commands included in the file to be obeyed can be further calls of OBEYFILE for other files. This process of nesting calls of OBEYFILE can continue to four levels. Note that for the second and subsequent levels the optional second parameter is ignored. This parameter is also ignored if OBEYFILE is called in background mode (see chapter 19).

The command OPTION

This command is used to set a number of optional characteristics for this user. The command takes one or more keyword parameters, which are listed below. The defaults are underlined.

The call only affects the ones that are specified, all other options in force being left alone (cf. PARM). The call does not take effect until after logging off and logging on again. It remains in effect until the end of the session in which the command OPTION is used again.

<u>QUICKSEARCH</u>	When searching for standard Subsystem commands, do not search the user's own library index or any library indexes appended (see chapter 10).
FULLSEARCH	When searching for commands, always search current library index and appended library indexes first.

<u>FULLMESSAGES</u>	Print confirmatory messages from commands (see the table in chapter 6).
NOMESSAGES	Suppress confirmatory messages.
<u>NORECALL</u>	Do not store interactive terminal input/output messages for use by RECALL (chapter 12).
TEMPRECALL	Store interactive terminal I/O for use by RECALL for the duration of the current session.
PERMRECALL	Store up to 16 pages (64K bytes) of the most recent interactive terminal I/O for use by RECALL. This is kept between sessions.
STACK=n	n can take an integer value in the range 2-8. This specifies the size (in segments of 64K bytes) to be used for the stack file created by the Subsystem, with the name 'SS#STK'. The default is 2 segments (128K bytes), which will be large enough for the majority of programs. See chapter 10 and reference 6 for the use of the stack.

The parameter ? can also be used, to cause the currently effective options to be printed out.

Examples:

```
OPTION(FULLSEARCH,STACK=4)
OPTION(FULLMESSAGES)
OPTION(?)
```

The command QUEUES

This command is used to determine the number of files belonging to this user which are held in EMAS queues awaiting output. It takes no parameter and output is of the form:

```
FILES QUEUED
LOCAL DEVICES LP CP   REMOTE NO: 37
NO OF FILES    3  1              2
```

In this case there are three files waiting to be printed on the line printer, one on the card punch and two waiting to be dispatched to remote terminal 37.

Notes

* Normally files are output on a particular device in order of submission.
* In the case of remote terminals the output from QUEUES makes no distinction between the devices available. Hence files for .CP37 and .LP37 and .P37 all appear as files for Remote number 37.

The command STOP

This command is used to terminate a foreground session. It takes no parameters. It has the following effects:

* Prints out usage information as for METER (see chapter 20).
* Destroys all temporary files created during this session and the default compiler listing file SS#LIST if it exists.
* Disconnects interactive terminal - making it available for another user.
* Stops the user's virtual processor - freeing a slot for another user to log on.
* Indicates to the demons process that any NOW jobs waiting for this user can be started as soon as machine time is available.

The command SUGGESTION

This command is provided to make it easy for users to send suggestions for changes or improvements to EMAS to the System Manager. Its use is intended primarily for minor items which crop up during an interactive session which do not merit the formality of a letter. The facility should not be used for reporting serious faults - these should be reported to the Advisory Service as soon as possible.

Users are warned that although an effort will be made to reply to all SUGGESTIONS eventually, they should not expect a prompt response. An indication will be given in the reply as to the likelihood of their suggestion being implemented.

The method of use is to type the command with no parameter. The replies to the prompts 'SURNAME:' and 'TEXT:' should be the personal (not EMAS) name of the user, and the text of the suggestion terminated with an asterisk. The example shows this:

```
        COMMAND: SUGGESTION
        SURNAME: JONES
        TEXT: TEXT OF SUGGESTION*
```

As many lines of text as desired may be given.

APPENDIX 1
SUBSYSTEM ERROR MESSAGES

Common error messages produced by the Subsystem should be self explanatory. The general ones are described more fully below, and those specific to an individual command are described in the part of the manual relating to the command.

GENERAL ERROR MESSAGES RELATING TO FILES

These messages can be produced following faults in many commands. The text '&&' in a message is replaced by the relevant name at the time of the failure.

ATTEMPT TO EXTEND ANOTHER USER'S FILE

If WRITE or WRITE SHARED access permission has been given a user may write to a file belonging to another user, but may not alter the size of the file.

CONFLICTING USE OF FILE && BY ANOTHER USER

This message will be produced when an attempt is made to connect a file in a way which is incompatible with the use currently being made of it by another user. Examples:

An attempt is made to compile into an object file belonging to this user which is permitted to another user in READ SHARED mode and is currently being executed by him.

An attempt is made to read from a file which belongs to another user (and is permitted to this user in READ SHARED mode) at the same time as the owner of the file is running a program which is writing to the file (see also chapter 3).

FILE && CURRENTLY BEING EXECUTED

This is only likely to occur when calling EMAS commands from within a program; it indicates that an attempt has been made to carry out some conflicting action on a file that is currently loaded, for example DESTROY it, open it for writing (see also chapter 18).

FILE && CURRENTLY IN USE

This message will be produced if an attempt is made to operate on a file in a way which is inconsistent with its present use; for example if an attempt is made to DESTROY a file which is currently being used for output (see also chapter 18).

FILE && CURRENTLY IN USE BY ANOTHER USER

This message is produced when an attempt is made to carry out a critical operation on a file belonging to this user which is permitted to, and currently connected in the virtual memory of, another user. Example: DESTROY, RENAME, OFFER.

FILE && DOES NOT EXIST

This message is produced when a user attempts to access a file belonging to himself which does not exist (or, exceptionally, has been permitted to self with an access permission of NONE).

FILE && DOES NOT EXIST OR NO ACCESS

For reasons of security when accessing files belonging to other users, no distinction is made between the fact that a file does not exist and the fact that it is not permitted to this user.

FILE INDEX FULL

This message indicates that there are no free file descriptors in the user's file index and an attempt has been made to create a file. Each user is allowed 32, 64 or 128 files (see chapter 3). The file creation may have been explicit, for example COPYFILE, or may be as a result of an attempt by the Subsystem to create a file in, for example, DETACH, FLIST(.LP) or LIST.

FILE INDEX FULL (NO FREE LIST CELLS)

A user's file index can be full when there are still free file descriptors, if all the list cells are in use. See chapter 3 for a full description of the use of list cells.

FILE SYSTEM FULL

The EMAS immediate file store is divided into a number of separate file system parts (currently 8, with 4 normally assigned to each machine). If the file system part containing this user becomes full and an attempt is made to create or extend a file this message will result.

INVALID FILENAME &&

A full file name is of the form:

```
        username.filename
```

where username contains 6 upper case letters or numerals and filename contains between 1 and 8 upper case letters or numerals, the first of which must be a letter. In addition, for files created by the Subsystem, the third character of the filename can be '#'. When referring to one's own files it is permissible to omit the username and the full stop. This failure message will be produced if a filename used as a parameter to a command does not conform to these rules.

INVALID OWNERNAME

If the ownername part of a filename is not the name of a user in one of the currently available file systems then this message will be produced. This can also occur with the commands OFFER and PERMITFILE. If the owner concerned is on the other machine then this may indicate that the other machine is temporarily unavailable.

REQUESTED ACCESS TO FILE && NOT PERMITTED

If a file is permitted to this user in READ or READ SHARED mode and an attempt is made to connect the file for writing - for example, EDIT(filename) - this message will be produced.

NO WRITE ACCESS TO FILE ON OTHER MACHINE

As explained in chapter 2 it is not possible to write to a file belonging to a user on the other machine, even though he has given appropriate access permission.

TOO MANY FILES CONNECTED

There is a limit on the number of files that can be connected in a user's virtual memory at any one time. Currently this limit is 94. If this fault occurs, files which are not currently needed should be disconnected, using the command DISCONNECT.

VIRTUAL MEMORY FULL

This fault, which is similar to the last one and which can be overcome in the same way, occurs when there is no room to connect a file in the virtual memory. For each virtual processor, the part of the virtual memory available to the Subsystem and to user programs and other user files is 13 Mbyte. In practice this fault has rarely occurred.

OTHER FAILURE MESSAGES

It is hoped that other messages produced by the Subsystem will be self explanatory when read in context. The production of clear diagnostic information both in relation to the Subsystem and to user program failures is regarded as being of importance. Further improvements are being made in this area. Users are encouraged to contact the Advisory Service if they have problems in this regard.

APPENDIX 2
CHARACTER CODES

Notes

* EMAS uses an internal character code (Table A) based on the 7 bit code for data interchange defined by the International Standards Organisation (ISO). The code assigns graphical or control characters to code values 0-127. Most of the control characters will only concern users of special terminals but they are included for completeness. The graphical representation of some characters will vary from one terminal to another. Common alternatives are shown in the table.

* Table B shows the mappings between card punchings and internal codes. It only contains those characters that can be punched directly on an IBM 029 Card Punch. The table uses the convention that the card rows are numbered from the top in the order 12 11 0 1 2 3 4 5 6 7 8 9.

Table A: EMAS Internal Character Code

0	NUL	32	space	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#(£)	67	C	99	c
4	EOT	36	$($\pi$)	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF(NL)	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\(~)(¬)	124	¦
29	GS	61	=	93	]	125	}
30	RS	62	>	94	↑(^)	126	‾
31	US	63	?	95	_	127	DEL

Table B: IBM 029 Card Code to Internal Code

Card Code	Internal Code	Character	Card Code	Internal Code	Character
No punching	32	Space	4-8	64	@
11-2-8	33	!	12-1	65	A
7-8	34	"	12-2	66	B
3-8	35	#	12-3	67	C
11-3-8	36	$(π)	12-4	68	D
0-4-8	37	%	12-5	69	E
12	38	&	12-6	70	F
5-8	39	'	12-7	71	G
12-5-8	40	(	12-8	72	H
11-5-8	41	)	12-9	73	I
11-4-8	42	*	11-1	74	J
12-6-8	43	+	11-2	75	K
0-3-8	44	,	11-3	76	L
11	45	-	11-4	77	M
12-3-8	46	.	11-5	78	N
0-1	47	/	11-6	79	O
0	48	0	11-7	80	P
1	49	1	11-8	81	Q
2	50	2	11-9	82	R
3	51	3	0-2	83	S
4	52	4	0-3	84	T
5	53	5	0-4	85	U
6	54	6	0-5	86	V
7	55	7	0-6	87	W
8	56	8	0-7	88	X
9	57	9	0-8	89	Y
2-8	58	:	0-9	90	Z
11-6-8	59	;	11-7-8	92	\(~)(¬)
12-4-8	60	<	0-5-8	95	_
6-8	61	=			
0-6-8	62	>			
0-7-8	63	?			

APPENDIX 3
GLOSSARY

This section provides brief definitions of a number of terms and abbreviations used in this manual and elsewhere in relation to EMAS. It should be appreciated that some of the terms do not have generally accepted definitions and so may be used for different purposes with respect to other systems.

Acoustic Coupler - A hardware device which can be used in some cases instead of a modem. It does not require the special wiring required for a modem.

Archive Store - Long-term store for users' files; held on magnetic tape (see chapter 3).

Backup Store - A magnetic tape copy of users' files currently held in the immediate store. Used in the event of a major system or disc file failure to replace damaged copies of files in the immediate store (see chapter 3).

Background Mode - A method of using EMAS without an interactive terminal. Background jobs are normally put in a queue by the user during a foreground session and are executed later - mainly overnight.

British Standard Interface (BSI) - A hardware Interface of standard design used to provide a fast link between computers of different manufacture. BSIs are used to connect the NCP to the two 4-75 computers and to the PDP15 computer (see chapter 2).

Byte - The smallest addressable unit in 4-75 main store (or in virtual store). Contains 8 binary bits. Used in IMP for variables of type %BYTEINTEGER and in FORTRAN for variables of type LOGICAL*1.

Core Store - Part of the hardware which contains the relevant program instructions and data whilst a program is being executed. In EMAS, user programs address core store via the paging mechanism. Also called main store.

Demons - The systems process which primarily handles file transfers to and from local peripheral devices (such as the line printer) and the Communications Network. It also checks usernames and passwords at log-on, and schedules batch jobs.

Digital Equipment Corporation (DEC) - The manufacturer of the main components of the NCP and TCPs, and of the Interactive Graphics Processor (PDP15).

Director - Part of the operating system, local to a particular process. It is paged and is contained in part of the same virtual memory as the Subsystem and user programs.

Disc file - A storage device on the 4-75 computer which holds a user's immediate files. There is one disc file for each 4-75, which is divided into four logical parts. Each 4-75 can access the disc file on the 'other' machine, for reading only (see chapter 2).

Dual Service - The term used when one 4-75 central processor is not available; all users and both disc files are assigned to the remaining 4-75.

File Header - An area at the beginning of a file that contains information about the type, size and, in the case of Data files, format of the file.

Foreground Mode - The method of using EMAS from an interactive terminal.

Front End Processor - This term, which is now falling into disuse, is used to refer to the NCP and those TCPs which are sited beside the 4-75 computers.

Hardware - The physical components of a computer, such as the central processor unit, the disc file etc.

Hashing - A method of storing names in a table in such a way that a particular name can be located efficiently.

Head Crash - A hardware fault usually affecting a disc file and corrupting or destroying some users' immediate files. It occurs when a recording head comes into contact with the surface of a disc, causing physical damage to the recording medium.

Hexidecimal - A number system using a radix of 16. The digits are represented by the characters 0-9 and A-F. It is a convenient system to use on an 8 bit byte machine, such as the 4-75, since the contents of each byte can be represented by two hexadecimal digits.

Immediate Store - The immediate store is held on the disc files of the 4-75 computers. It contains all the user files which are currently available to active users.

Initial Program Load (IPL) - Strictly a hardware function involving reading in a minimal control program to an otherwise empty machine. In practice this term covers the whole process of initialising the 4-75 computers for an EMAS session. The process includes a number of hardware tests of critical devices and a consistency check of the immediate file store indexes.

Interactive Terminal - The general name for a terminal used for accessing EMAS in foreground mode - i.e. interactively. Can be a Teletype, VDU or some other form of terminal. Also called a console.

International Computers Ltd. (ICL) - The manufacturers of the 4-75 computers on which EMAS is run.

Modem - A hardware device connected between an interactive terminal and a telephone circuit to convert signals from the terminal into a form suitable for transmission over the telephone circuit, and vice versa.

Network Control Processor (NCP) - A small computer (DEC PDP 11/45) which links the two 4-75 computers to each other and to the TCPs. Also provides link to Regional Communications Network.

Operating System - A suite of programs that controls usage of the computer; in EMAS this includes Supervisor, Director, Subsystem and System Processes.

Page - The smallest unit (4096 bytes) of file space allocation. Main store is allocated to virtual processors in pages and information is transferred between main store and drums and disc files in units of pages. See also Units of Storage.

Page Turn - A unit used in the accounts algorithm. A page turn is charged to a user process each time a page is copied for the process between core and a disc file or drum. A user can minimise the number of page turns by organising his data efficiently but it is unlikely to have much effect except with very large programs.

Paging - A method of accessing main store by dividing it into page units and addressing them through a translation mechanism. When used in conjunction with backing store of drums or disc files, it allows many processes with varying storage requirements to use a limited main store as if each had a large allocation of contiguous store.

PDP15 - The computer connected to EMAS primarily for interactive graphics applications. Also used for activities independent of EMAS. Manufactured by DEC.

Phase Encoded (PE) - A mode of storing data on magnetic tape. Tapes written on EMAS are all PE, as against NRZ (Non Return to Zero) mode.

Program Loading - The process of preparing for an object file to be executed, which entails connecting the file in virtual memory and satisfying any external references it makes. See chapter 10.

Regional Communications Network - A number of computers and remote job entry terminals mainly in Edinburgh and Glasgow (including the EMAS NCP) which are connected together in a network.

Segment - A unit of file space, equal to 16 pages or 64K bytes or 65,536 bytes.

Segmentation - The division of virtual memory into segments. A file is always connected at a segment boundary (even though it may be more than 1 segment long). This simplifies paging by limiting the range of addresses at which a file may be connected in different virtual memories. Not of direct concern to user - rather for the convenience of the supervisor.

Software - A general name for programs as distinct from hardware. Also used for operating systems components (as against user and applications programs).

Supervisor - A component of EMAS. It is resident in main store all the time, i.e. not paged. It contains code for handling peripherals, for example drums and disc file, allocating resources to virtual processors and moving pages to and from main store.

System File Information (SFI) - The part of a user's file index which contains information that is required to be retained between sessions - such as OPTIONs (see chapter 21).

System Process - A process which carries out certain system functions rather than running user programs. It runs in a virtual processor using a virtual memory, as does a user process, and is scheduled in a similar way by the resident supervisor. For example, the demons process is a system process which is responsible for transferring files to and from peripherals such as the line printer.

System 4-75 - The largest computer in the ICL System 4 range. Byte addressed, paged machine with order code similar to the IBM 360 range. EMAS is run on a twin 4-75 configuration.

Terminal - A hardware device used for communicating with a computer. It may be an interactive terminal such as a Teletype or a remote computer used for job entry or line printer output.

Terminal Control Processor (TCP) - A small computer used to connect a group of interactive terminals via the NCP to the 4-75s. Some of the TCPs are connected directly to the NCP, others are at remote locations and are connected by communication lines. Most TCPs are PDP11/10 computers.

Units of Storage - The table below shows the relationship between the main units used for measuring storage:

	Byte	Kbyte	Page	Segment
Mbyte	1048576	1024	256	16
Segment	65536	64	16	
Page	4096	4		
Kbyte	1024			

Virtual Address - An address by which a program accesses a location in the main store. The virtual address is converted by the paging mechanism into a real address in the main store. The IMP function ADDR returns the virtual address of a variable.

Virtual Memory (VM) - An addressable area which appears to be a contiguous area of up to 16 Mbytes. It is mapped by the paging mechanism onto pages in the main store or on drum or disc file. Of the 16 Mbytes in a complete virtual memory 13 Mbytes are available to the Subsystem and user programs and files.

Visual Display Unit (VDU) - An interactive terminal which displays its output on a cathode ray tube rather than using paper.

REFERENCES

The following publications are referred to in this manual. Copies of them all are available for reference in the ERCC Library. Some are available for purchase at the same location.

1. 'Edinburgh IMP Language Manual' - Editor R. McLeod, ERCC 1974
2. 'Edinburgh FORTRAN Language Manual' - Editor W. Aitken, ERCC 1976
3. 'Edinburgh ALGOL Language Manual' - Mrs F. Stephens, ERCC 1976
4. 'The EMAS Scheduling and Allocation Procedures in the Resident Supervisor' - N.A. Shelness, P.D. Stephens and H. Whitfield, Department of Computer Science and ERCC, University of Edinburgh 1974
5. 'EMAS The Edinburgh Multi-Access System' - H. Whitfield and A. Wight, The Computer Journal, Vol No 4 November 1973
6. '4-70 Series Processors' - ICL 1972 Technical Publication 1104
7. 'System 4 Hardware Reference Peripherals' - ICL 1970 Technical Publication 4505
8. 'ERCC Graphics Manual' - Editor J. Murison, ERCC 1976
9. 'The Edinburgh IMP/FORTRAN System Library Manual' - Editor Miss L. Carlton, ERCC 1974
10. 'O.S. User's Guide' - Editor J. Murison, ERCC 1976
11. 'A Syntactic and Semantic definition of the IMP Language' - P.D. Stephens, ERCC 1974
12. 'O.S. Tape Labels' - IBM, Form No. GC28-6680

INDEX